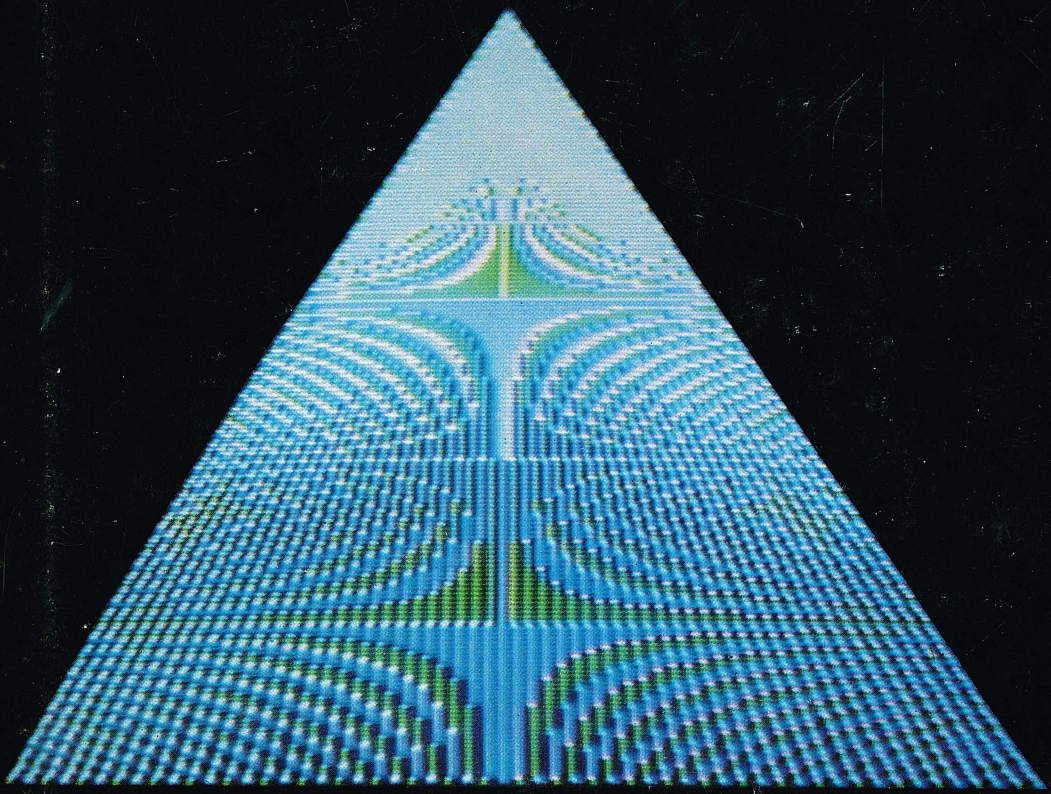
Computer Literacy Skills

GRAPHICS AND ANIMATION ON THE APPLE

II, II+, IIe, and IIc



CHRISTOPHER LAMPTON

GRAPHICS AND ANIMATION ON THE APPLE

II, II+, IIe, and IIc

BY CHRISTOPHER LAMPTON

A COMPUTER
LITERACY SKILLS BOOK

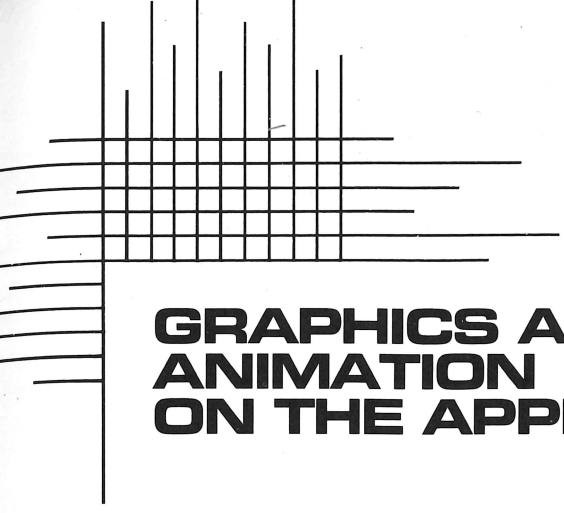
The words *computer graphics* and *animation* conjure up magical images of arcade games, bright colors, flashing lights, and wonderful shapes.

This book shows you how to create graphics on computers in the Apple II series, which are among the most popular computers used at home and in schools. Some knowledge of the BASIC language is needed, but you needn't be an expert programmer.

You will learn several ways to create graphics using the computer's low- and high-resolution modes. Numerous programs and activities give you a chance to try and experiment with nearly every technique presented. Each chapter concludes with a group of suggested projects for further exploration.

So why depend on commercial programs to make your life more graphic and animated? Chris Lampton will show you how to create your own games, designs, and charts.

FRANKLIN WATTS
387 PARK AVENUE SOUTH
NEW YORK, NEW YORK 10016



GRAPHICS AND ANIMATION ON THE APPLE

GRAPHICS AND ANIMATION ON THE APPLE

II, II+, IIe, and IIc

By Christopher Lampton

A Computer Literacy Skills Book
FRANKLIN WATTS 1986
New York London Toronto Sydney

Original designs created by Grant Geyer,
Marshall Geyer, and Jonathan Halpern.
Photographs courtesy of: Data Trek: pp. 2, 50, 79;
Ira Schulman: pp. 9, 23, 27, 56, 65, 66, 70, 78;
Brøderbund: pp. 46, 59, 65, 72; Electronic Arts:
pp. 48, 53; Apple Computer, Inc.: p. 32.

Library of Congress Cataloging in Publication Data

Lampton, Christopher.
Graphics and animation on the Apple.

(A Computer literacy skills book)
Bibliography: p.
Includes index.
Summary: Explains how to use the computer language
BASIC to program the Apple II series computers to
create graphics. Includes suggested projects.
1. Apple II (Computer)—Programming—Juvenile
literature. 2. Computer graphics—Juvenile literature.
[1. Apple II (Computer)—Programming. 2. BASIC
(Computer program language) 3. Programming (Computers)
4. Computer graphics] I. Title. II. Series.
QA76.8.A662L354 1986 006.6'765 85-26613
ISBN 0-531-10143-6

Copyright © 1986 by Christopher Lampton
All rights reserved
Printed in the United States of America
6 5 4 3 2 1

CONTENTS

CHAPTER ONE
Getting the Picture 1

CHAPTER TWO
Formatting the Screen 5

CHAPTER THREE
Low-Resolution Graphics 13

CHAPTER FOUR
Low-Resolution Applications 31

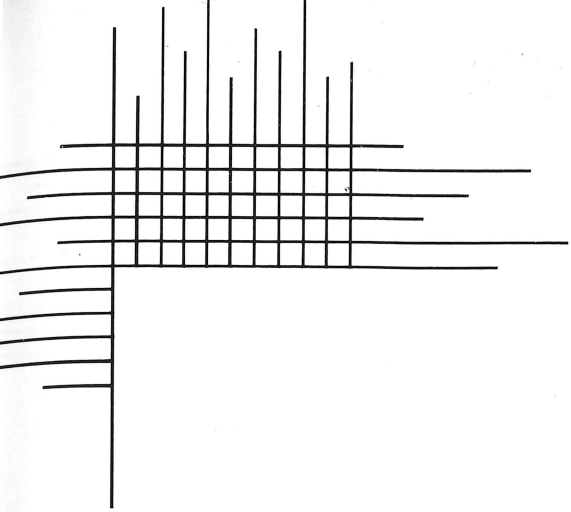
CHAPTER FIVE
High-Resolution Mode 45

CHAPTER SIX
High-Resolution Lines 63

CHAPTER SEVEN
High-Resolution Shapes 81

Bibliography 95

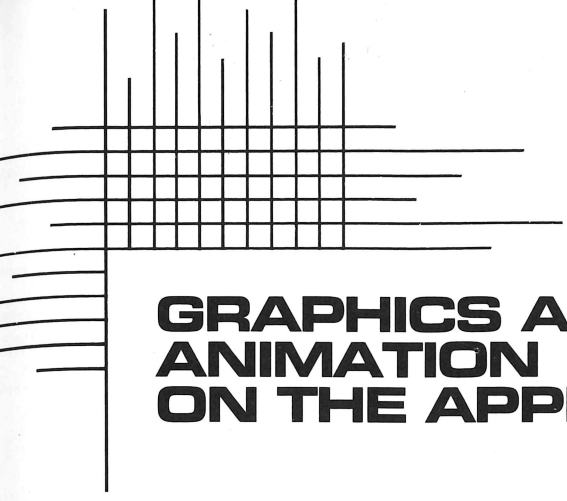
Index 97



OTHER BOOKS BY
CHRISTOPHER LAMPTON

Advanced BASIC
BASIC for Beginners
Black Holes and Other Secrets of the Universe
COBOL for Beginners
Computer Languages
Dinosaurs and the Age of Reptiles
FORTH for Beginners
FORTRAN for Beginners
Fusion: The Eternal Flame
Graphics and Animation on the Atari
Graphics and Animation on the Commodore 64
Graphics and Animation on the TRS-80
Meteorology: An Introduction
PASCAL for Beginners
PILOT for Beginners
Planet Earth
Prehistoric Animals
Programming in BASIC
6502 Assembly-Language Programming
Space Sciences
The Sun
Z80 Assembly-Language Programming

The publisher wishes to acknowledge the assistance of Marshall Geyer, Grant Geyer, and Jonathan Halpern in the preparation of this book.



GRAPHICS AND ANIMATION ON THE APPLE

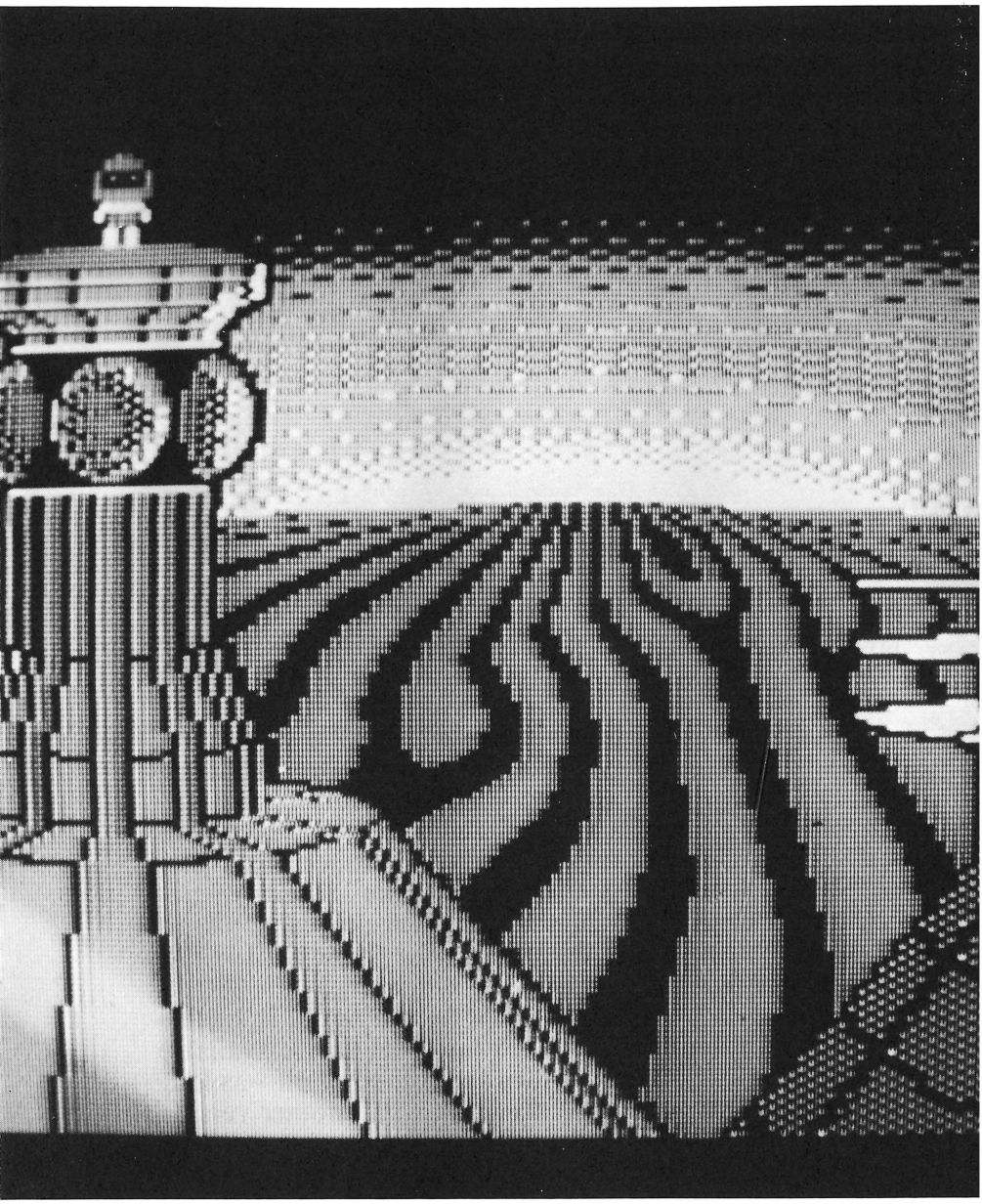
1 GETTING THE PICTURE

If a picture is worth a thousand words, then a computer graphic, which can move, change colors, interact with an observer, and do many other things an ordinary picture cannot, must be worth a thousand pictures.

This book is about computer graphics, pictures "drawn" on the video display of a computer. Although some of the principles that we will discuss here are applicable to any computer capable of graphics, we will be specifically discussing the ways a relatively inexperienced programmer can create graphics on an Apple II series microcomputer using the programming language called Applesoft BASIC. The Apple II series includes the Apple II, II+, IIe, and IIc computers.

All of these computers, except for the Apple II, have a built-in version of the Applesoft language that is available from the moment that you turn the computer on—as long as you don't have a disk in the drive with another version of BASIC on it. If you have an early Apple computer with only the Integer BASIC language installed, you can buy a version of Applesoft on disk. Some of the programs in this book will also work in Integer BASIC, but many of them will not.

If you have access to any of these computers, you will be able to sit down at the keyboard, type the programs pre-



Willy Byte in the Digital Dimension,
designed by Murray Krehbiel and
Greg Hammond and sold by Data Trek.

sented in this book, and watch your computer produce pictures. Better still, if you have some smattering of knowledge about how to write programs in the computer language BASIC (only a rudimentary knowledge is required), you will be able to use the information in this book to create graphics programs of your own.

The Apple II was the first popular home computer capable of producing high-resolution, color graphics, that is, detailed multicolor images on a video display. Although other computers have come along since that can be used to create sophisticated color graphics, the Apple still holds its own in this regard. The ability of the Apple to produce high-resolution color images is equal, or almost equal, to that of any other computer on the market.

PIXELS

Just as all matter is made up of atoms, all computer pictures are made up of *pixels,* or *pictorial elements.* A pixel is simply a tiny rectangle of color on the computer's video display. By mixing together pixels of various colors, a computer programmer can produce pictures of just about anything, in the same way that a newspaper produces black-and-white pictures on paper by mixing together tiny dots of varying sizes. If you've never done so before, pick up a newspaper and look closely at any photograph, preferably through a magnifying glass. You'll see that each image is made up of what photographers call halftone dots. When viewed closely, the image seems like little more than a collection of black dots; when viewed at a distance, it becomes a remarkably vivid picture.

The Apple II series computers feature both high- and low-resolution graphics. High-resolution graphics use a very small pixel, rather like the halftone dots in the newspaper photo, so that we can pack a lot of details into our pictures. Low-resolution graphics use a much larger pixel, so that we can put much less detail into our pictures.

Why would we want to work with low-resolution graphics at all? For one thing, they are easier to use than high-resolution graphics, because there are generally fewer pixels to worry about. Furthermore, more colors are available when we use low-resolution graphics. The Apple allows us to create low-resolution pictures using a palette of sixteen

colors, while only six colors are available for high resolution. And those colors aren't quite as easy to use, as you shall see much later in this book.

Are you ready to start drawing pictures on your Apple? Good. Sit down at the computer, prepare it for programming in BASIC, then turn to the next chapter . . . where we will begin.

2 FORMATTING THE SCREEN

When you first run the Applesoft BASIC interpreter, the video display will be in *text mode.* This means that we can put various characters on the display, such as letters of the alphabet, numerals, punctuation marks, and so forth. In fact, these characters will begin appearing from the moment you turn the computer on, when the words APPLE II appear at the top of the display. You can make more characters appear by typing them on the Apple keyboard.

You can also write computer *programs* that make characters appear on the video display. As you already know, if you are familiar with Applesoft BASIC (or any version of BASIC, for that matter), the instruction PRINT can be used to place individual characters, and strings of individual characters, on the display. For instance, the instruction

```
PRINT "THIS IS A STRING OF CHARACTERS!"
```

will make the sentence THIS IS A STRING OF CHARACTERS appear on the display, if you type it while the BASIC interpreter is running and press the RETURN key at the end of the instruction. The programming term *string* simply refers to the fact that several characters are "strung"

together into a larger group; the computer treats this string as a single unit, and will dutifully print character after character onto the display, until it reaches the end.

(In this book, we will print only uppercase letters on the display. If you have an Apple computer that will print lowercase letters as well, you may change uppercase letters to lowercase as may seem appropriate. We will occasionally use lowercase letters in REM statements, to make them more readable, but you need not transcribe these to your computer if you do not wish to do so; or you may convert the lowercase letters to uppercase, if necessary.)

In a sense, we are creating pictures when we print characters on the video display of the computer: pictures of the characters. These are not the kind of picture that you ordinarily think of when you hear the term "computer graphics," but if you look closely you will see that these text characters are indeed made up of tiny dots. And, yes, these dots are pixels. Some other computers have special characters called graphics characters that can be used to create larger pictures, though the Apple does not. We cannot create pictures of anything other than these text characters as long as the Apple is in text mode.

In a moment, we will see how other kinds of graphics can be created through Applesoft BASIC programs. For the moment however, let us see what we can do using merely the text capabilities of the Apple. As we saw above, we can print a string of text characters on the display with the PRINT statement. And, of course, we can incorporate such PRINT statements directly into an Applesoft BASIC program, simply by adding program lines, like this:

```
10 PRINT "THIS IS A STRING OF CHARACTERS!"
```

By adding a second line to this short program:

```
20 GOTO 10
```

then typing RUN and pressing RETURN, we can fill the screen with words. A pretty picture indeed!

The normal Apple text screen can display twenty-four lines of forty characters each. Thus, we say the Apple has a forty-*column* screen, because it holds forty columns of text characters. The Apple IIc and most Apple IIe's, as well as

many specially adapted earlier models, can also display eighty columns of characters; however, in this book we will concentrate only on the forty-column capabilities of the Apple text mode.

HTAB AND VTAB

If we are interested in more precise formatting of our words, we can use the Applesoft commands HOME, VTAB, and HTAB to determine the precise position at which text strings are printed on the display. This is called *cursor positioning.* Although you may think of the *cursor* as the tiny symbol that marks the position at which your next typed character will appear, programmers use this term to indicate the position on the screen where the next string of characters will begin, when printed by the PRINT command.

The HOME command moves the cursor to the "home" position, in the upper left-hand corner of the display. It also clears the text screen. Type HOME in the BASIC immediate mode (that is, without a line number and followed by a carriage return). The screen will clear immediately, and the cursor will reappear at the top.

HTAB is short for "horizontal tab." It moves the cursor a specified number of columns horizontally from the left margin. The instruction HTAB 24, for instance, moves the cursor to column 24. The first column on the display is column 1.

Similarly, the VTAB command is short for "vertical tab." It moves the cursor a specified number of lines vertically, from the top of the screen. The instruction VTAB 5, for instance, moves the cursor to line 5 (where the first line of the display is line 1).

To print the word HELLO at column 17 on line 11, we could write and RUN the following program:

```
10 HOME
20 HTAB 17
30 VTAB 11
40 PRINT "HELLO"
```

The HOME instruction in the first line assures that the HELLO is printed on an otherwise blank screen. And the

HTAB and VTAB instructions position the cursor properly before the actual printing takes place.

INVERSE TEXT

Applesoft BASIC permits a number of interesting visual effects using only text characters. For instance, text may be reversed so that the characters appear as black on white rather than white on black. This is done with the INVERSE statement. After execution of the INVERSE statement, all characters are printed in reverse colors, until a NORMAL instruction is executed. The NORMAL instruction restores all subsequent text to a "normal" (i.e., noninverse) state. To print the words "REVERSED TEXT" on the display in inverse mode, we could write this program:

```
10 INVERSE
20 PRINT "REVERSED TEXT"
30 NORMAL
```

The NORMAL statement returns the display to normal.

The FLASH instruction causes characters on the display to flash back and forth between inverse and normal mode. Once a FLASH statement has been executed, every character printed subsequently will be in "flash mode," until a NORMAL statement is executed.

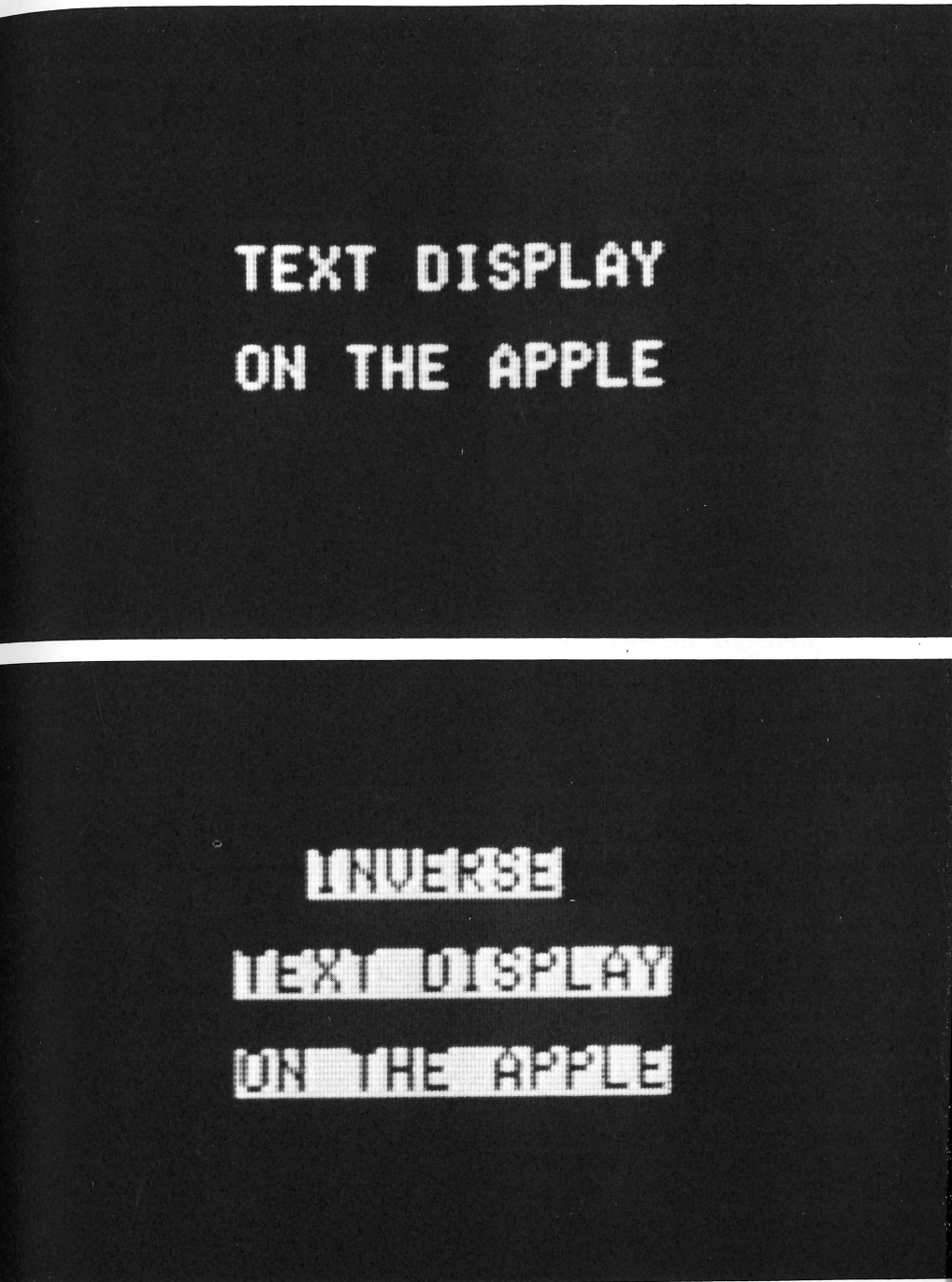
To print the words FLASHING TEXT in flashing characters, we could write

```
10 FLASH
20 PRINT "FLASHING TEXT"
30 NORMAL
```

This program mixes all of the modes together on the same display:

```
10 HOME
20 VTAB 5
```

Normal text (above) and reversed text (below)


TEXT DISPLAY
ON THE APPLE
INVERSE
TEXT DISPLAY
ON THE APPLE

```
30 HTAB 15
40 INVERSE
50 PRINT "PRESENTING"
60 NORMAL
70 VTAB 10
80 HTAB 8
90 PRINT "THE FABULOUS, INCREDIBLE"
100 VTAB 15
110 HTAB 9
120 FLASH
130 PRINT "APPLESOFT TEXT DISPLAY"
140 NORMAL
150 GOTO 150
```

Type this program, then type RUN and press RETURN. When you have finished looking at the display, press CONTROL-C (that is, the CONTROL key and the C key simultaneously) to stop the program from executing.

Although this book is about graphics, we will sometimes want to put text and graphics together on the same display. The Apple's ability to mix text and graphics is limited, but when we do mix text and graphics, we will be using the text display and formatting instructions that we have studied in this chapter.

In the next chapter, we will study the Apple low-resolution graphics mode, and see how real graphics are produced.

Suggested Projects

1. Write a program that writes the word HELLO on the display twenty-four times, in a column down the center of the display.

2. Write a program that writes the word HELLO on the display, with every other letter in flash mode.

3. Write a program, using the Applesoft TEXT mode, that turns the entire screen of the computer, except for the border, white. (**Hint:** Use the INVERSE command.)

3
LOW-RESOLUTION GRAPHICS

Like most computers with graphics capabilities, the Apple II series computers make a distinction between a text display and a graphics display. Although these two different types of display will both appear on the same video monitor, and may seem superficially similar, the computer will treat them in a very different manner.

A text display is used for displaying text, predefined characters that usually represent the letters of the alphabet, numerals, punctuation marks, etc. The graphics display is used for displaying . . . well . . . graphics, which is to say pixels. In general, we cannot have text and graphics on the same display, without indulging in fancy trickery (such as drawing text characters on the display as though they were pictures), though the Apple does allow us to create a "split screen" that is part graphics display and part text display.

The Apple II series also makes a distinction between the low-resolution graphics display and the high-resolution graphics display. The most obvious difference between the two is in the size of the pixels with which drawings can be created. However, there will be less obvious differences between them as well, including differences in the commands we use to create pictures on these displays in Applesoft BASIC.

THE GR STATEMENT

To turn on the Apple's low-resolution display, you have to use the Applesoft instruction GR. To exit the low-resolution display and return to the text display, you use the Applesoft instruction TEXT. These instructions can be typed in the immediate mode or used in a program.

To see what happens when the low-resolution display is turned on, type GR in the immediate mode and press RETURN. You will probably notice two things. One is that the screen is cleared. The other is that you can still type characters on the last few lines of the screen. This is the split screen effect mentioned a few paragraphs ago. Applesoft BASIC will not place the entire screen in graphics mode when we issue the GR command; it reserves the bottom four lines as a *text window,* so that we can display ordinary text characters along with our graphics. As you shall see, this can be quite useful.

Nonetheless, there will be times when you will want to eliminate this text window altogether. Fortunately, there is a way to do this. You can type

```
POKE −16302,0
```

either in the immediate mode (followed by a RETURN) or in a program.

Unfortunately, when you do this you will see some strange, colored stripes at the bottom of the screen after the instruction is executed. In this book, we will simply leave these stripes on the screen when we create a full screen low resolution graphics display. If you'd prefer, however, you can get rid of the text window *and* the stripes by using the following multiple instruction, either in the immediate mode or in a program:

```
POKE −16302,0 : COLOR = 0 : FOR X = 0 TO 39 : FOR Y =
40 TO 47 : PLOT X,Y : NEXT Y : NEXT X
```

A second GR command will restore the split screen.

Since most of the programs that follow do not use the bottom lines of the low-resolution screen, you may also delete the POKE −16302,0 instruction from these programs without causing harm. Even if the programs do make use of

the bottom portion of the low-resolution screen, deleting this instruction will not cause any errors in program execution, though strange characters may appear in the text window.

How does POKE −16302,0 work? The POKE command places the number 0 at a special location in the Apple's *memory* (the series of electronic circuits inside the computer that stores programs, text, graphics data, etc.). This tells the Apple that no text window is desired and will put the computer into a full-screen low-resolution graphics mode. Be warned, however, that if you type this command in the immediate mode, you will no longer be able to see what you are typing, because there will be no more text window for your typing to appear in. (You *will* see funny graphics images appear at the bottom of the screen while you type, but they won't be readable. These funny images are the computer's vain attempt to interpret your typing as graphics information, that is, as a picture.) The computer will still understand what you are typing, and will obey any commands that you type if they are terminated with RETURN, but you will have to type blind, without being able to see any mistakes you might make. For now, we will leave the text window on.

COLOR AND PLOT

How do we draw pictures on the low-resolution display? For starters, we can use the COLOR and PLOT statements. The COLOR statement tells the computer what color we want to draw with. The Apple gives us a choice of sixteen different colors in low-resolution mode. Each of these colors is identified by a number, according to the chart below:

Black	- 0	Dark Green	- 8
Red	- 1	Orange	- 9
Dark Blue	- 2	Gray #2	- 10
Purple	- 3	Pink	- 11
Blue	- 4	Green	- 12
Gray #1	- 5	Light Green	- 13
Medium Blue	- 6	Aqua	- 14
Light Blue	- 7	White	- 15

Before using a PLOT statement to draw pixels on the display, always use the COLOR instruction to tell the Applesoft interpreter what color you plan to draw with. The COLOR instruction is used like this:

```
COLOR = n
```

where n is a number between 0 and 15, representing one of the sixteen colors just listed. This number can be represented by a variable, or as the result of an arithmetic calculation such as X + 3, if you wish. The number represents the color to be used for drawing.

For instance, if you wished to draw with green, you would type:

```
COLOR = 12
```

Type this command in the immediate mode and press RETURN. Nothing visible will happen—not right away, that is.

For the most part, we will create drawings on the low-resolution display one pixel at a time, by changing the color of pixels at specific positions on the video display. This is done with the PLOT statement. We use the PLOT statement like this:

```
PLOT x,y
```

where x and y are the *coordinates* of the pixel whose color we wish to change.

What do we mean by coordinates? The first coordinate, called the *horizontal coordinate*, tells the computer how many pixels from the left side of the display the desired pixel is located. The leftmost pixel on the display has a horizontal coordinate of 0. Since the low-resolution display is forty pixels wide, the rightmost pixel has a horizontal coordinate of 39. Thus, the horizontal coordinate in a PLOT statement can have any value from 0 to 39.

The second coordinate, called the *vertical coordinate*, tells us how many pixels from the top of the display the desired pixel is located. The topmost pixel on the display has a vertical coordinate of 0. Since the low-resolution dis-

play is forty pixels tall, the bottommost pixel on the display has a vertical coordinate of 39 in split-screen mode. The display is forty-eight pixels tall in full-screen mode. Thus, the vertical coordinate in a PLOT statement can have any value from 0 to 39 when the display is in split-screen mode and any value from 0 to 47 when in full-screen mode.

Suppose we wanted to display a pixel in the center of the screen in green. We would need to use two commands, like this:

```
COLOR = 12
PLOT 20,20
```

Type these two instructions in the immediate mode, pressing RETURN after each. (Type a GR instruction first, if your display is not still in the low-resolution mode. If you haven't turned off the machine or issued additional COLOR instructions since you typed the COLOR = 12 instruction above, there is no need to type it again.) A green pixel should appear roughly in the middle of the display, against the black background. The first statement, COLOR = 12, told the computer that we wanted to draw in green. The PLOT 20,20 statement told the computer that we wanted to display the pixel at coordinates 20,20—roughly the center of the display—in the current plotting color.

We can turn this pixel back off, that is, make it disappear, by plotting it to the background color. Since the background color is normally black, we must first issue a COLOR = 0 statement, then PLOT the pixel again with the PLOT 20,20 statement. Here is a short program that will blink the pixel on and off:

```
10 GR : HOME
20 COLOR = 12
30 PLOT 20,20
40 FOR X = 1 TO 500
50 NEXT X
60 COLOR = 0
70 PLOT 20,20
80 FOR X = 1 TO 500
90 NEXT X
100 GOTO 20
```

Type this program and RUN it. The FOR-NEXT loops in lines 40 to 50 and 80 to 90 do nothing except slow the program down, so that the pixel won't blink too quickly.

DRAWING LINES

Ordinarily, we will want to turn on more than one pixel. Here is a program that uses a FOR-NEXT loop to draw a horizontal line across the middle of the display:

```
10 GR : HOME
20 POKE -16302,0
30 COLOR = 15
40 FOR X = 0 TO 39
50 PLOT X,20
60 NEXT X
70 GET A$
80 TEXT
90 HOME
```

Type and RUN this program.

Notice that the first two lines turn on a full-screen graphics mode. The third-to-last line (70) pauses the program until any key is pressed. Line 80 restores the normal text mode. Also notice that we use a HOME command immediately after the GR instruction; this is because characters left on the TEXT screen can sometimes cause clutter at the bottom of the low-resolution graphics screen, as the computer attempts to interpret the text as graphics information. The GR command automatically clears the top portion of the screen, but it leaves the text window portion untouched, even when we use a POKE command to remove the window.

Do you see how the line was drawn? The FOR-NEXT loop that begins in line 40 of the program steps variable X through all of the possible values of the horizontal coordinate of a PLOT statement. Line 50 then PLOTs those coordinates, with an unchanging vertical coordinate of 20. Thus, a horizontal line is drawn from one side of the display to the other, at vertical coordinate 20.

In the same way, we can draw a vertical line from the top of the display to the bottom by PLOTting at all possible vertical coordinates, but leaving the horizontal coordinate unchanged. Here is a program that does precisely that:

```
10 GR : HOME
20 POKE -16302,0
30 COLOR = 15
40 FOR Y = 0 TO 47
50 PLOT 20,Y
60 NEXT Y
70 GET A$
80 TEXT
```

HLIN AND VLIN

As it happens, all of the work done by the last two programs is irrelevant (except that they illustrate certain graphics programming principles), since Applesoft BASIC features two commands intended specifically for drawing horizontal and vertical lines: HLIN and VLIN. As you might guess, the first draws horizontal lines and the second draws vertical lines.

We use the HLIN command like this:

```
HLIN x1,x2 AT y
```

where x1 is the horizontal coordinate of the top end of the line, x2 is the horizontal coordinate of the bottom end of the line, and y is the vertical coordinate of the entire line. The first number (x1, above) must always be smaller than the second number (x2). Like the horizontal coordinates in a PLOT statement, these two numbers must always be in the range 0 to 39. And the third number (y) must be in the vertical coordinate range of 0 to 39 (or 47, in full-screen mode). The color the line is drawn in is determined by a COLOR statement, just like a PLOTted pixel.

Here is a program that utilizes HLIN to draw a horizontal line:

```
10 GR : HOME
20 POKE -16302,0
30 COLOR = 15
40 HLIN 0,39 AT 20
50 GET A$
60 TEXT
```

We use the VLIN command like this:

```
VLIN y1,y2 AT x
```

where y1 is the vertical coordinate of the top end of the line, y2 is the vertical coordinate of the bottom end of the line, and x is the horizontal coordinate of the entire line. The first number (y1, above) must always be smaller than the second number (y2). Like the vertical coordinates in a PLOT statement, these two numbers must always be in the range 0 to 39. And the third number (x) must be in the horizontal coordinate range of 0 to 39 (or 47). The color the line is drawn in is determined by a COLOR statement, just like a PLOTted pixel.

Here is a program that utilizes VLIN to draw a vertical line:

```
10 GR : HOME
20 POKE -16302,0
30 COLOR = 15
40 VLIN 0,47 AT 24
50 GET A$
60 TEXT
70 HOME
```

Of course, there are types of lines that cannot be drawn by HLIN and VLIN, and so must be drawn with a series of PLOT statements, or a single PLOT statement strategically placed in a program loop. Here, for instance, is a program that draws a diagonal line across the display:

```
10 GR : HOME
20 POKE -16302,0
30 COLOR = 15
40 X = 0
50 FOR Y = 0 TO 39
60 PLOT X,Y
70 X = X + 1
80 NEXT Y
90 GET A$
100 TEXT
110 HOME
```

DIAGONAL LINES

To draw a diagonal line, the values of both the horizontal and vertical coordinates must be changed. One of the coor-

dinates must always be increased by 1 (and no more) for each pixel in the line., The other pixel must be increased by a value no greater than 1 for each pixel. To change the angle of the line, change the value by which the second coordinate is increased. For instance, in the program above, you can change line 70 to read

```
70 X = X + .5
```

and the angle of the line will become steeper.

How can we change the coordinate of a pixel by 0.5? Actually, we can't. The Applesoft interpreter ignores any fractional value in a coordinate and rounds the value of that coordinate to the next smallest whole number. However, the fractional value *is* stored in the variable that we are using to represent the coordinate, and when the next whole number in the sequence is reached, the coordinate changes. Thus, the coordinate changes only every other time the loop is executed, if we add 0.5 to a coordinate. If we add 0.25 to the coordinate, it will change only once every fourth time the loop executes, and the angle of the line will become even steeper. You can experiment with other diagonal lines, using different values to increase the horizontal coordinate. Also try increasing the value of the horizontal coordinate with the FOR-NEXT loop, and adding fractional values to the vertical coordinate.

This short subroutine will draw a line between *any* two points on the low-resolution display:

```
5000 DX = X2 - X1 : DY = Y2 - Y1
5010 IF ABS(DX) < ABS(DY) THEN 5080
5020 YI = DY/ABS(DX)
5030 FOR X = X1 TO X2 STEP SGN(DX)
5040 Y = Y1 : PLOT X,Y
5050 Y1 = Y1 + YI
5060 NEXT
5070 RETURN
5080 XI = DX/ABS(DY)
5090 FOR Y = Y1 TO Y2 STEP SGN(DY)
5100 X = X1 : PLOT X,Y
5110 X1 = X1 + XI
5120 NEXT
5130 RETURN
```

When this subroutine is called, variables X1 and Y1 should contain the horizontal and vertical coordinates, respectively, of one end of the line. Variables X2 and Y2 should contain the horizontal and vertical coordinates of the other end of the line. The subroutine doesn't check to make sure these coordinates fall into the legal range, so be careful that you don't ask it to draw a line that runs off the edge of the display.

You can call the subroutine like this:

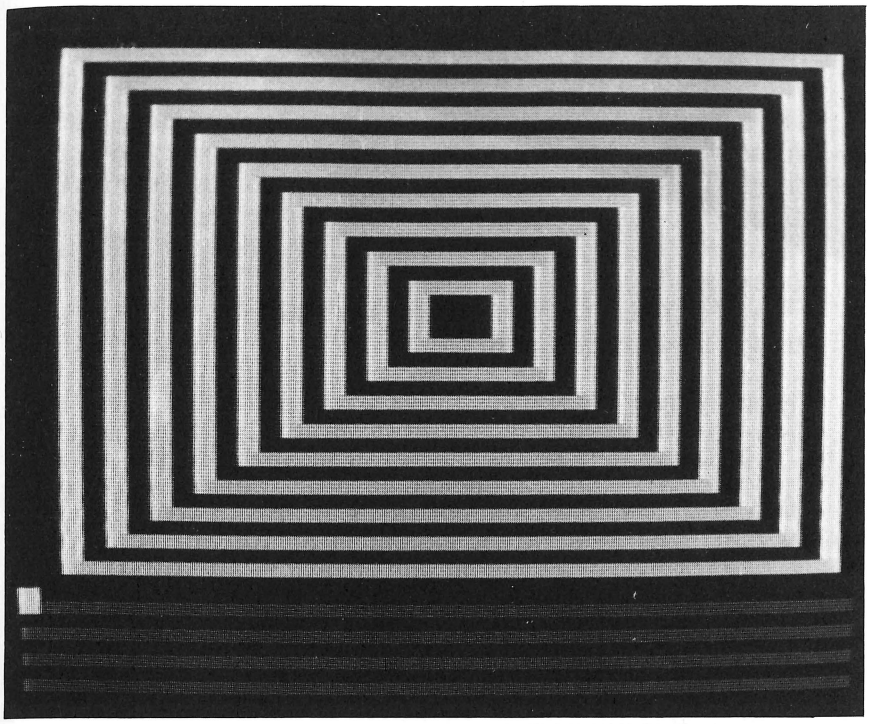
```
10 GR : HOME
20 COLOR = 15
30 POKE -16302,0
40 X1 = 10 : Y1 = 12 : X2 = 38 : Y2 = 28
50 GOSUB 5000
60 GET A$
70 TEXT
80 HOME
90 END
```

Type this short program, add the subroutine above, and run the resulting program. You will see a line drawn from coordinates 10,12 to coordinates 38,28. Press any key to return to text mode.

DRAWING SHAPES

We can use the graphics commands that we have studied so far to draw various shapes on the low-resolution screen. A rectangle is easy; it is simply a combination of lines. Here is a program that uses HLIN and VLIN instructions to draw a rectangle in the middle of the display:

```
10 GR : HOME
20 POKE -16302,0
30 COLOR = 15
40 HLIN 10,30 AT 10
50 HLIN 10,30 AT 30
60 VLIN 10,30 AT 10
70 VLIN 10,30 AT 30
80 GET A$
90 TEXT
100 HOME
```



Design created from the program below

Here is a slightly trickier program that draws a series of concentric rectangles in the middle of the display:

```
10 GR : HOME
20 POKE -16302,0
30 COLOR = 15
40 FOR W = 4 TO 36 STEP 4
50 GOSUB 1000
60 NEXT W
70 GET A$
80 TEXT
90 HOME
100 END
1000 HLIN 20 - W/2, 20 + W/2 AT 20 - W/2
1010 HLIN 20 - W/2, 20 + W/2 AT 20 + W/2
1020 VLIN 20 + W/2, 20 - W/2 AT 20 + W/2
```

```
1030 VLIN 20 + W/2, 20 - W/2 AT 20 - W/2
1040 RETURN
```

The subroutine at line 1000 draws a rectangle in the center of the display with a height and width of W, measured in pixels. The FOR-NEXT loop in lines 30 to 50 increments (that is, increases) the value of W from 4 to 36, with a step value of 4. RUN the program, then press any key to return to text mode.

We can also use the HLIN and VLIN instructions to draw a filled rectangle, like this:

```
10 GR : HOME
20 POKE -16302,0
30 COLOR = 15
40 FOR X = 10 TO 30
50 VLIN 10,30 AT X
60 NEXT X
70 GET A$
80 TEXT
90 HOME
```

Type this program and RUN it. The FOR-NEXT loop in lines 30 to 50 will step the horizontal coordinate of the vertical line drawn in program line 40 through all values between 10 and 30, thus drawing a series of vertical lines across the screen, forming a filled box. By using the HLIN instruction, and stepping the vertical coordinate with the loop, we could draw the box from top to bottom, rather than from left to right. And by giving the loop a STEP value of -1, we could reverse the direction of drawing, if this should be desirable.

By enlarging the "box" to the size of the screen, we change the color of the display background from black to any of the other fifteen low-resolution colors. For instance, if we wanted to draw a gray line on a blue background, we could write:

```
10 GR : HOME
20 COLOR = 4
30 FOR X = 0 TO 39
40 VLIN 0,39 AT X
50 NEXT X
```

```
60 COLOR = 5
70 HLIN 0,39 AT 20
80 HOME
90 GET A$
100 TEXT
110 HOME
```

By using this method to recolor the background, we can plot any color over any other color.

The following program draws a series of twenty boxes on the display, in random colors and sizes:

```
10 GR : HOME
20 FOR I = 1 TO 20
30 X = RND(1) * 32
40 Y = RND(1) * 32
50 HW = RND(1) * (32 - X) + 4
60 VW = RND(1) * (32 - Y) + 4
70 COLOR = RND(1) * 15 + 1
80 GOSUB 1000
90 NEXT I
100 END
1000 FOR X1 = X TO X + HW - 1
1010 VLIN Y, Y + VW - 1 AT X1
1020 NEXT X1
1030 RETURN
```

This program (which you may type and RUN) makes extensive use of the Applesoft RND function. If you are not familiar with the RND function, you may want to take a quick peek at your Applesoft BASIC manual to see how it works. Briefly, when used with an argument (the number after RND in parentheses) of 1 (or any other positive number), it generates a random fraction between 0 and 1 (but never equal to 1). By multiplying RND (1) by another number, we can produce a random number between 0 and that other number (but not including that other number). RND(1) * 98, for instance, will generate a random number between 0 and 98, but not including 98.

In the box-drawing program, we have used the RND function to produce both the horizontal coordinate (variable X) and vertical coordinate (Y) of the upper left-hand corner of each box, as well as the horizontal width (HW)

and the vertical width (VW) of each box. (The vertical and horizontal widths cannot be smaller than 4.) Line 70 uses RND to generate a random color number between 1 and 15. (This is done by generating a random number between 0 and 14 with RND(1) * 15, then adding 1 to it. Fractional values in these numbers are ignored by the COLOR, VLIN, HLIN, and PLOT statements.)

You can make several changes to this program to make it more interesting. For instance, the number of boxes generated can be increased (or decreased) by changing the limit of the loop in line 30. You can cause the program to loop back and generate a new set of boxes every time a key is pressed, by changing line 100 to

```
100 GOTO 10
```

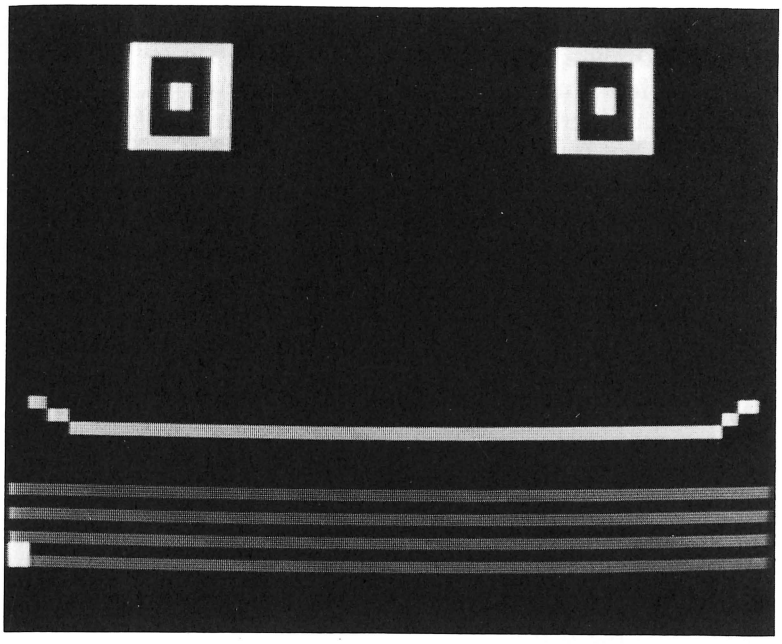
You can generate screen after screen of boxes automatically by changing line 100 to

```
100 GOTO 20
```

Now that you know how to plot pixels and draw lines on the low-resolution screen, try writing a few programs of your own. Draw random lines using the line-drawing subroutine we showed you earlier. Or try drawing pictures. Using the Applesoft GET statement, which will set a string variable equal to the value of the last key pressed, write a program that allows you to draw lines on the screen simply by pressing keys. (The W key might move the line upward, the X key might move it down, the A key could move it left, the D key could move it right, and so on. Be sure to check so that the line won't run off the edge of the screen.)

ANIMATION

We need go only a short distance from drawing pictures on the low-resolution screen to making those pictures move, that is, animating them. Although the Applesoft low-resolution graphics commands are not ideal tools for creating computer animation, we can make some limited gestures in that direction. For instance, here is a program that moves a pixel from one side of the display to the other:



Greg Geyer, Marshall Geyer, and Jonathan Halpern—students attending school in Staten Island, New York—created this design using techniques described in this chapter.

```
10 GR : HOME
20 POKE -16302,0
30 FOR X = 0 TO 39
40 COLOR = 15
50 PLOT X,20
60 COLOR = 0
70 PLOT X,20
80 NEXT X
90 GET A$
100 TEXT
110 HOME
```

The pixel generated above is a little flickery, but it does seem to move. Observant readers will note that this pro-

gram is almost identical to the program earlier in this chapter that drew a horizontal line across the display, except that we have now added a second pair of COLOR and PLOT statements (and moved the original COLOR statement so that it is inside the FOR-NEXT loop). The second PLOT statement now rePLOTs the pixel to the background color (COLOR 0), effectively erasing it. Instead of seeing a line drawn across the display, we see what appears to be a lone pixel racing along a horizontal path.

Although this program is rather crude, it illustrates one of the primary principles of computer animation: draw a picture in one position, erase it, then draw it again in another position. Sophisticated arcade games for the Apple have been written using this seemingly simple method of animation.

We'll study animation in more detail later. First, we'll consider some more "serious" matters.

In the next chapter, we'll look at a couple of low-resolution graphics applications, one of them useful and the other just plain fun.

Suggested Projects

1. Write a program that draws a line from the upper right-hand corner of the display to the lower left-hand corner.

2. Write a program that prompts the user for the height and width of a square, then draws a square to those dimensions.

3. Write a program that draws a succession of vertical stripes across the display, in each of the sixteen low-resolution colors. You may use colors more than once. Rewrite the program to draw horizontal stripes across the display.

4. Write a program that moves an animated pixel from the top of the display to the bottom. Write a program that allows the user to guide an animated pixel back and forth on the display by pressing keys on the keyboard of the computer.

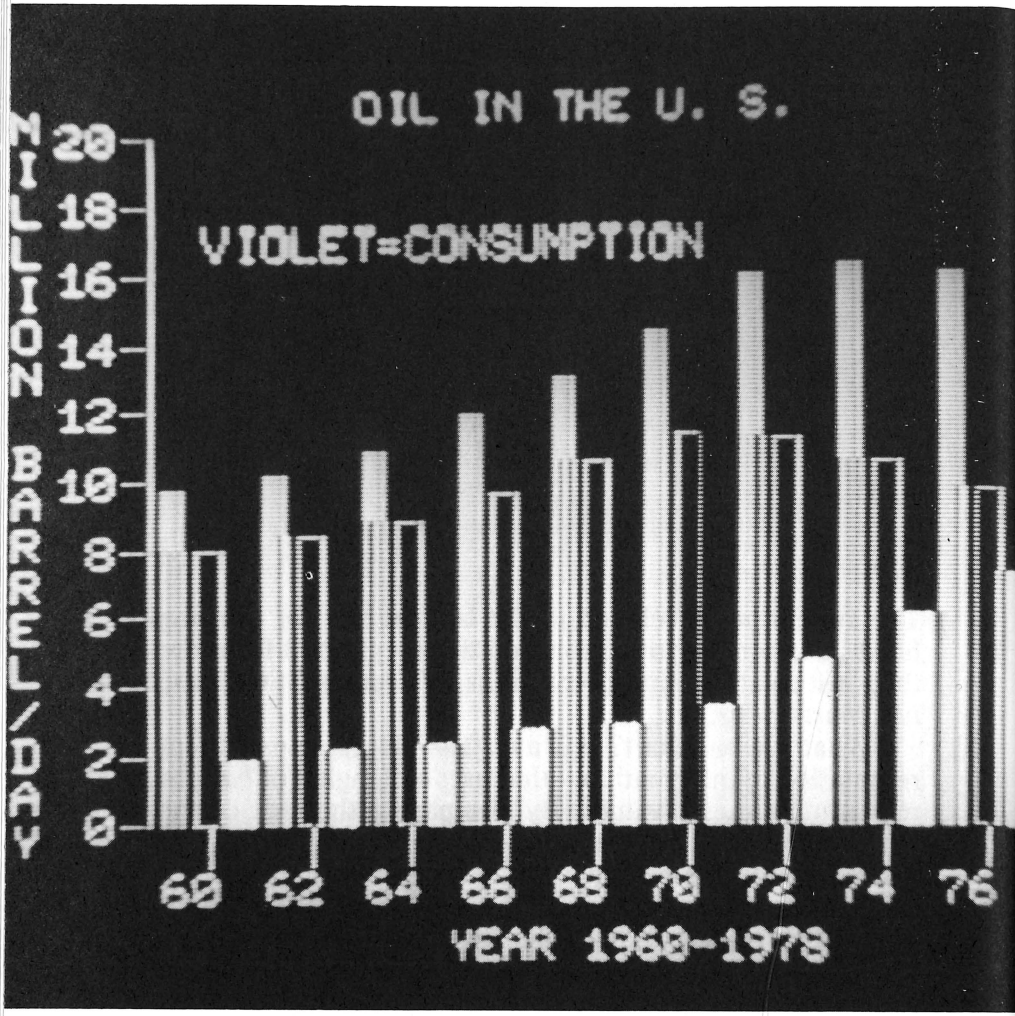
4
LOW-RESOLUTION APPLICATIONS

The system we developed in the last chapter for drawing filled boxes on the low-resolution display is good for more than just creating party decorations. We can use it to create *bar charts.*

What is a bar chart? It is a chart—that is, a visual representation of information—that uses rectangles, or *bars*, to represent numeric values. By comparing the size of the bars, we can visually compare the size of the values. A large bar represents a large value, while a small bar represents a small value. Usually, it is either the height or the width of the bar that represents the value, depending on whether we want the bars to grow upward, toward the top of the video display (or the paper, if we are actually drawing the chart with pen or crayon), or sideways, toward the left- or right-hand side of the display (or paper).

You might, for instance, use a bar chart to represent how much you have earned at a part-time job, with each bar in the chart representing one week at the job and the size of the bar representing the dollar amount that you earned that week. A large bar would represent a large amount of earnings for that week; a small bar would represent a small amount of earnings.

A computer like the Apple is an excellent medium for creating bar charts. Not only can it perform all the neces-



Commercially created bar chart

sary calculations about the sizes of the bars used to represent various values, but it can actually draw the bars as well, using the low-resolution display. If you have a printer attached to your computer that can print graphics, the computer can then draw a copy of the chart on paper.

In the first part of this chapter, we'll create a bar chart program in Applesoft BASIC. It won't have many frills, such as the ability to print the bar charts on a printer, or store the bar chart data on a disk, but you can expand it to do these things, if you wish. What it *will* do is calculate the size of the bars, based on the information you give it, and then draw the chart. First it will ask questions about the chart to be drawn. Here is the first question it will ask:

```
WHAT IS NAME OF CHART?
```

When the chart is actually drawn, the name will be printed at the bottom of the chart. Because we can't print the name on the low-resolution display, we will have to use a split screen, with a text window at the bottom.

Here's the second question the program will ask:

```
HOW MANY BARS?
```

This is the actual number of bars in the chart. Because we are working in low resolution, we must place a fairly strict limit on the number of bars to be drawn. Rather arbitrarily, we will set this limit at 12. If the user of the program responds with a number larger than 12, the question will be repeated. Since a bar chart with fewer than two bars would be meaningless, we will also set a lower limit of two bars.

Here's the third question the program will ask:

```
WHAT IS STANDARD UNIT?
```

The standard unit is a tricky concept. It is the unit that the values represented by the bars are measured in. For instance, if you are graphing your earnings at a part-time job (as suggested above), the standard unit might be the *dollar.* Alternatively, it could be the *cent,* if your earnings are *very* small. Or, if your earnings are very, very large, you might even use *tens of dollars* or *hundreds of dollars* as the standard unit.

The actual standard unit used in a given chart is insignificant; all that matters is that you use the same unit for all the bars in a single chart. If you earned 50 dollars in each of two weeks, you couldn't tell the bar chart program that you earned 50 dollars for one week and 5,000 cents for the other. Although this would be perfectly true, the program would draw a much larger bar for the second week than for the first, despite the fact that you earned the same amount in each. When the bar chart is printed, the standard unit used will be indicated in the text window at the bottom.

The program will then ask the user how large each bar should be, in standard units. It will query the user separately for each bar, like this:

```
LENGTH OF BAR #1 (IN STANDARD UNITS)?
```

Once the program has recorded the length of each bar, it is ready to draw the chart. However, it must make certain decisions before the chart can be drawn. The size specified by the user for each bar is in standard units, and this standard unit is chosen by the user, according to the type of information to be represented by the chart. However, when the actual bars are drawn, they will be measured in a different unit: the pixel. So, in order to determine how long each bar will be in pixels, the program must first decide how many pixels it takes to represent a standard unit.

This isn't as tough as it sounds. If we use the variable SU to represent the length of the standard unit in pixels, we can calculate that length like this:

```
SU = SH/LB
```

SH is the height of the area of the screen reserved for drawing bars, measured in pixels. Since we need to reserve a segment of the screen for the text window, we will set SH equal to 30 in the program, though later we will see how this value can easily be changed to accommodate higher-resolution charts. LB is the length of the largest bar, in standard units. This length can be determined by the program itself while the lengths of the bars are being input.

The program must also decide on the width of the individual bars, in pixels. This is based purely on the number of

bars requested, and the width of the display in pixels. If we use the variable W to represent the width of a bar in pixels, we can calculate that width like this:

```
W = SW/NB
```

SW is the width of the display in pixels—40, in the case of the Apple low-resolution display. NB is the number of bars in the chart, which is determined by the third question that the program asks of the user.

Once the program has determined all of these values, it is a relatively simple matter to draw the rest of the chart. Here is the program:

```
10 MB = 10 : SW = 40 : SH = 39 : REM MB = MAXIMUM
NUMBER OF BARS, SW = SCREEN WIDTH (IN PIXELS), SH =
SCREEN HEIGHT (IN PIXELS)
20 DIM L(MB) : REM SET UP NUMERIC ARRAY TO HOLD
LENGTHS OF 10 BARS
25 LB = 0 : REM LB = LONGEST BAR
30 HOME : TEXT : REM CLEAR SCREEN
40 INPUT "WHAT IS NAME OF CHART?"; T$
50 IF LEN(T$) > 30 THEN PRINT "NAME TOO LONG" : PRINT
"LESS THAN 30 CHARACTERS, PLEASE" : GOTO 40
60 PRINT "HOW MANY BARS? (2-";MB;")"; : INPUT NB
70 IF NB < 2 OR NB > MB THEN PRINT "OUT OF RANGE" :
GOTO 60
80 INPUT "WHAT IS STANDARD UNIT? "; SU$
90 IF LEN(SU$) > 30 THEN PRINT "NAME TOO LONG" : PRINT
"LESS THAN 30 CHARACTERS, PLEASE" : GOTO 80
100 FOR I = 0 TO NB - 1 : REM LOOP FOR NB BARS
110 PRINT "LENGTH OF BAR #"; I+1; " IN STANDARD
UNITS?";
120 INPUT L(I)
130 IF L(I) < 1 THEN PRINT "MUST BE GREATER THAN
ZERO" : GOTO 100
140 IF LB < L(I) THEN LB = L(I) : REM GET LENGTH OF
LONGEST BAR
150 NEXT I
160 HOME : GR : REM ENTER LOW-RESOLUTION MODE
170 COLOR = 15
180 W = SW/NB : REM CALCULATE WIDTH OF BARS
```

```
190 SU = SH/LB : REM CALCULATE # OF PIXELS IN
STANDARD UNIT
200 VTAB 22% : HTAB 18 - LEN(T$)/2 : PRINT "**"; : INVERSE
: PRINT T$; : NORMAL : PRINT "**" : REM PRINT TITLE
210 VTAB 23 : HTAB 16 - LEN(SU$)/2 : PRINT "(UNIT =";
SU$;")": REM PRINT STANDARD UNIT
220 FOR I = 0 TO NB - 1 : REM LOOP TO DRAW NB BARS
230 FOR Y = SH TO SH-SU*L(I) STEP -1 : REM DRAW EACH
LINE OF BAR
240 HLIN I*W, I*W+W-2 AT Y : REM DRAW ONE LINE OF BAR
250 NEXT Y : NEXT I
260 IF PEEK(-16384) < 128 THEN 260 : REM WAIT FOR KEY
PRESS (SEE SECOND HALF OF CHAPTER FOR EXPLANATION)
270 GOTO 25 : REM DO IT AGAIN
```

The program has been commented so that you can see how it performs its tricks. Type it and RUN it. It will pause after drawing each chart and wait for you to press a key, then it will request data for a brand-new chart. Feel free to add to the program as you wish. Some suggestions for expanding the program are included at the end of the chapter.

LOW-RESOLUTION GAMES

All work and no play makes the Apple a dull computer. We can also use low-resolution graphics, with the help of the animation technique we discussed in the last chapter, to create an arcade game.

Admittedly, we're not likely to duplicate one of the truly high-resolution arcade games, like *Donkey Kong* or *Tron.* The low-resolution display will not allow us to show sufficient detail, and programs written in Applesoft BASIC are just too slow, because of the time required for the interpreter to translate each program statement into a form that the computer can execute. Later, we will learn how to create high-resolution graphics, but Applesoft BASIC will still be too slow a medium for most arcade games (though you might want to apply some ingenuity to find some ways around that slowness).

Still, some simpler, low-resolution arcade games are ideal for low-resolution BASIC programming. In fact, one of the first popular games for the Apple computer, way

back in the ancient days of the 1970s, was called *Little Brick Out* and was based on the early arcade game *Breakout* (which can still be found in various home computer versions); in fact, you might have a copy of this game on one of the disks that came with your computer. In that game, the low-resolution screen was used to depict a brick wall, which the player demolished brick by brick with a paddle and a bouncing ball. (If you have to ask how a paddle and a bouncing ball can be used to demolish a row of bricks, then you haven't played many arcade games.) It is an excellent example of what can be done with low-resolution graphics on the Apple II.

In this chapter, we'll use the Apple low-resolution screen to create a variation on the classic microcomputer Ping-Pong game, where a Ping-Pong ball (actually, a pixel) is batted around the screen by a player with a paddle (actually, a row of pixels controlled by pressing keys on the keyboard). Although the best microcomputer Ping-Pong games allow two players to go head to head, we'll create a one-player variation.

There are two reasons for this. One is that it is easier to write a one-player game. The second is that our program will periodically need to check to see if the player is pressing a key on the keyboard, so that the player's paddle can be moved appropriately. Alas, there is no way to check to see if two keys are being pressed simultaneously, as would be the case if two players were playing against one another. (If your Apple computer is equipped to use a joystick, you may want to rewrite the game so that two players can compete using a pair of joysticks.)

The game will work like this: When the program is run, a colored border will be drawn around three sides of the screen. A small paddle, drawn as a straight vertical line of pixels, will appear at the left-hand side of the display. The paddle will move up and down in response to input from the keyboard. A moving ball will appear in the middle of the screen. When the ball hits one of the borders around the screen, or the paddle, it will bounce. If it goes off the unbordered side of the screen, and isn't intercepted by the paddle, the ball will be lost. The game will continue as long as the player continues to hit the ball; points will be awarded every time the paddle connects with the ball.

Drawing the playfield will be simple. The three-sided

border can be depicted with HLIN and VLIN instructions; similarly, the paddle can be sketched with a single VLIN statement. The ball, which will be a single pixel, can be drawn with a PLOT statement.

We will also need to do two things we have not done in earlier programs: check a given coordinate position on the low-resolution screen to see if it is already occupied by a pixel of a particular color, so that we will know when the ball collides with the border or the paddle; and check to see if a key has been pressed on the keyboard without halting the flow of the program.

The first—checking a coordinate position for a colored pixel—can be done easily enough with the SCRN function. You use the SCRN function like this:

```
SCRN(x,y)
```

where x and y are the horizontal and vertical coordinates of the low-resolution pixel that we want to check. The SCRN function will automatically be set equal to the color value of the pixel at that coordinate. Thus, if the pixel at coordinate 15, 27 is purple in color, the statement

```
A = SCRN(15,27)
```

will cause the variable A to be set equal to 3, which is the color value for purple. If we make the border and the paddle yellow, we can test to see if the ball is about to collide with the wall by writing

```
T = SCRN(X,Y)
```

where X and Y are the coordinates of the position the ball is *about* to occupy, and checking to see if T is equal to 13, the color code for yellow. If not, we can assume the ball is sailing through open space. If so, we'll assume a collision, and reverse the direction of the ball to create a "bounce" effect.

Checking the keyboard without halting the flow of the program is somewhat more difficult. There are only two commands for reading the keyboard in Applesoft BASIC. One is INPUT, which pauses the execution of the program until the RETURN key is pressed. Not an ideal way for our

program to check to see if a key is being pressed; we cannot halt the movement of the ball while the player decides which way to move the paddle. The GET command is somewhat closer to what we need, but it also halts the flow of the program until a key is pressed.

The answer to this problem is to avoid Applesoft commands altogether and go straight to the source by examining the memory location within the Apple that *always* contains a code number representing the last key pressed. (This is the memory location that Applesoft itself uses to determine what keys you are pressing.) The memory location in question has an *address* of −16384. (We'll discuss the concept of memory addresses in more detail in Chapter Seven.) You can examine this location using the PEEK function.

You use the PEEK function like this:

```
PEEK(x)
```

where x is the address within the computer's memory that you wish to examine. PEEK will be set equal to the numeric value, between 0 and 255, currently stored at that address.

The instruction

```
K = PEEK(−16384)
```

will set the variable K equal to the *ASCII code* of the key currently being pressed, plus 128. The ASCII code, or *American Standard Code for Information Interchange*, is a coding system that uses numbers between 0 and 127 to stand for letters of the alphabet, numerals, punctuation marks, and so forth. If no key is being pressed, then this location will hold a number less than 128. To find out the ASCII codes generated by the keys on your keyboard, run this short program:

```
10 K = PEEK(−16384)
20 IF K < 128 THEN 10
30 PRINT K − 128
40 GOTO 10
```

The keys that we will use to move the paddle are W and X. If the W key is being pressed, address −16384 will con-

tain the number 215. If the X key is being pressed, it will contain the number 216. Thus, to move the paddle, we will repeatedly PEEK memory to see if one of these two keys is being pressed. If the W key is being pressed, we will move the paddle up one pixel; if the X key is being pressed, we will move the paddle down one pixel. Immediately after we have PEEKed memory location −16384 and found that a key has been pressed, we must issue this instruction:

```
POKE −16368,0
```

This has the peculiar effect of erasing the value at location −16384. (Well, actually it resets the value to a number smaller than 128.) We have to do this; otherwise, the program would continue detecting the fact that the key had been pressed long after you had taken your finger off the key, and the movement of the paddle would become nearly uncontrollable.

Here is the Ping-Pong program:

```
10 GR
20 COLOR = 13
30 PY = 17 : BX = 10 : BY = 10 : DX = 1 : DY = −1 : DP = 0
: S = 0
35 REM PY = VERTICAL COORDINATE OF PADDLE, BX =
HORIZONTAL COORDINATE OF BALL, BY = VERTICAL
COORDINATE OF BALL, DX = HORIZONTAL DIRECTION OF
BALL, DY = VERTICAL DIRECTION OF BALL, DP = VERTICAL
DIRECTION OF PADDLE, S = SCORE
40 HLIN 2,39 AT 0
50 HLIN 2,39 AT 39
60 VLIN 0,39 AT 39
100 COLOR = 0
110 VLIN PY, PY + 4 AT 2 : REM ERASE PADDLE
120 PY = PY + DP : REM MOVE PADDLE
130 DP = 0 : REM HALT PADDLE
140 COLOR = 13
150 VLIN PY, PY + 4 AT 2 : REM REDRAW PADDLE
160 COLOR = 0
170 PLOT BX,BY : REM ERASE BALL
180 BX = BX + DX : REM MOVE BALL
190 BY = BY + DY
200 COLOR = 13
```

```
210 PLOT BX,BY : REM REDRAW BALL
220 K = PEEK(-16384) : IF K < 128 THEN 260 : REM IF NO
KEY PRESSED, SKIP
230 POKE -16368,0 : REM ELSE CLEAR KEY
240 IF K = 215 THEN DP = -1 : REM IF W PRESSED, MOVE
PADDLE UP
250 IF K = 216 THEN DP = 1 : REM IF X PRESSED, MOVE
PADDLE DOWN
260 IF (PY + DP < 0) OR (PY + DP > 34) THEN DP = 0 : REM
IF PADDLE OFFSCREEN, DON'T MOVE
270 IF SCRN(BX + DX, BY) = 13 THEN DX = -DX : REM IF
COLLISION IN HORIZONTAL DIRECTION, REVERSE DX
280 IF SCRN(BX, BY + DY) = 13 THEN DY = -DY : REM IF
COLLISION IN VERTICAL DIRECTION, REVERSE DY
290 IF BX = 3 AND DX = 1 THEN S = S + 1 : REM IF JUST
BOUNCED OFF PADDLE, INCREASE SCORE
300 IF BX + DX > 2 THEN 100 : REM IF BALL STILL ON
SCREEN, REPEAT
310 TEXT : HOME : REM ELSE GO TO TEXT SCREEN
320 PRINT "YOU MISSED!"
330 PRINT "YOUR SCORE IS ";S
```

ENHANCING THE GAME

Once again, the program has been documented as thoroughly as possible, so that you can see how it works. The key to the motion of the ball is in variables DX and DY, which contain the direction the ball is headed in. At every pass through the program loop, the value of DX is added to the horizontal coordinate of the ball (contained in BX) and the value of DY is added to the vertical coordinate of the ball (contained in BY). Because these variables are always equal to either 1 or −1, the ball will move along the horizontal and vertical coordinates simultaneously; thus, it will always seem to move on a diagonal. We make the ball bounce by reversing the value of either DX or DY (i.e., changing a value of 1 to −1, or vice versa).

After you've RUN the program, played with it, and figured out what makes it tick, you might want to modify it. As with the earlier bar chart program, this program is lacking in frills and could use some work.

For instance, the game play is terminated the moment the player misses the ball. You could rewrite the game so

that the player will be given a new ball every time he or she misses the ball, up to a certain maximum number of balls. Furthermore, the game score could be displayed in a text window at the bottom of the screen, rather than being presented to the player only at the end of play. Of course, if you plan to add a text window to the display, remember to move the bottom border of the screen above the window, and make sure that the paddle cannot run off the visible edge of the screen.

Another flaw in this program is that the ball starts at the same position on the display each time you play the game, and it is always heading in the same direction. Thus, the ball moves through exactly the same sequence of movements in each game; there is no sense of randomness. You might want to use the Applesoft RND function to select a random starting position and direction for the ball, to replace the fixed starting position and direction established in line 30 of this program. (The starting position of the ball is held in variables BX—horizontal coordinate—and BY—vertical coordinate. The direction is held in DX and DY.)

And once you have worked the problems out of this program, you might try your hand at a game of your own. It could be either a variation on the Ping-Pong game or a completely different kind of game. The only requirement is that the game must use low-resolution graphics and not make any great speed demands on the BASIC interpreter.

And if you find that low-resolution graphics are too limiting and that you really need to control fine details on the Apple display—well, the Apple also offers a high-resolution graphics mode, as we shall see in the next chapter.

Suggested Projects

1. Write a modified version of the barchart program presented in this chapter. Some features you might want to add include:

   a. Multicolored bars to represent different types of information. For instance, if you wanted to create a chart comparing the income that you have made with the income that a friend has made, you can use different colored bars for your income and for your friend's income.

   b. An option for storing the description of the chart (i.e., the information about the size and number of individual bars) in a file on a floppy disk, plus an option for reading this data back off the disk to re-create the chart on command.

   c. An option for drawing the chart on paper, using a printer with graphics capabilities.

2. Rewrite the Ping-Pong program in this chapter. Some features you might want to add include:

a. An option for using joysticks rather than the keyboard. If your Apple computer has cursor movement keys, you might also want to rewrite the program to use the cursor keys rather than the W and X keys.

b. An option for two players to go head to head, using joysticks.

c. A continuous display of the score in the text window.

d. An option for playing the game again after it ends.

e. Multiple Ping-Pong balls in each game, either sequentially, or (an interesting option) at the same time.

5
HIGH-RESOLUTION MODE

The Apple II series computers are justly famous for their high-resolution graphics. They deserve to be. As we noted earlier, the Apple II was the first popular personal computer to offer high-resolution graphics, and it thereby set the standard for the industry. Even today, some of the best-looking graphics-oriented software is available for the Apple II, despite the existence of several newer computers with graphics of comparable quality. Although it may no longer represent the state of the art in microcomputer graphics, the Apple II series can still hold its own.

And yet we have spent nearly the first half of this book discussing the Apple low-resolution mode. Many computers can create presentable low-resolution graphics. Why don't we just get on with it and show the reader how to program the sort of graphics that the Apple does best?

Well, we will . . . we will. Only, there's a problem. Apple II high-resolution graphics are a powerful medium for the creation of pictures, but they are not very easy to use. They are especially not easy to use in Applesoft BASIC. The truth is, the best Apple graphics programs are written in machine language—and that method of programming is far beyond the scope of the book.

So, our discussion of high-resolution graphics will be, by necessity, rather limited. To create pictorial master-

This was designed using DazzleDraw, a program manufactured by Brøderbund.

pieces, electronic Mona Lisas, on the Apple II display, you will have to buy special software. Fortunately such software is available in abundance, at fairly reasonable prices. If all you want to do is draw pictures, then you are better off buying a software package that will help you to draw pictures than you are trying to write picture-drawing programs in Applesoft BASIC.

And you won't learn how to create high-speed, high-resolution arcade games because, once again, BASIC is too slow to animate pictures of any complexity. Low-resolution graphics are a much better medium for this kind of game.

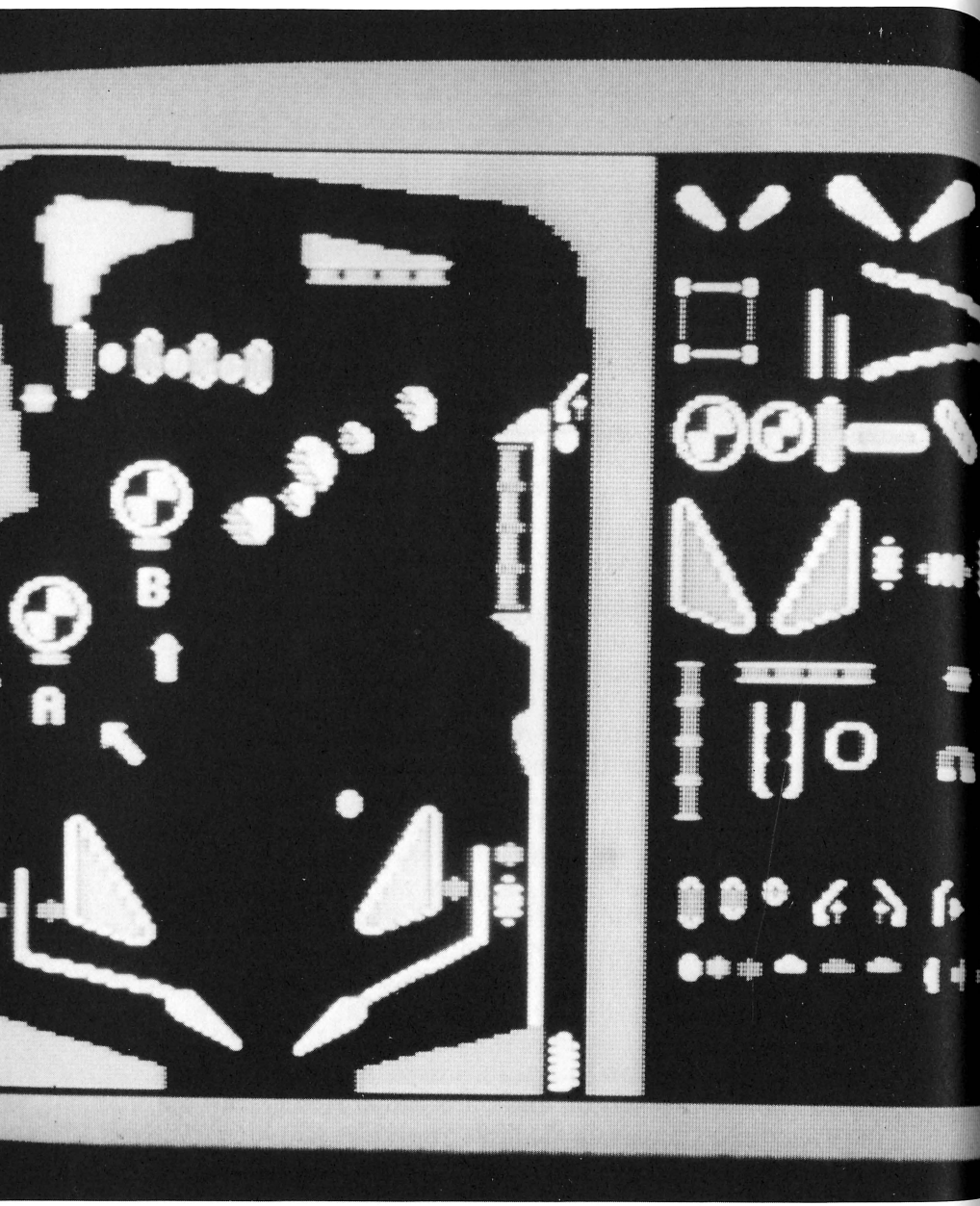
Nonetheless, you will learn how to create high-resolution pictures and how to do some useful things with the Apple high-resolution mode. And, with a little imagination, you may even use this information to do things that are very hard to do. If so, more power to you!

THE HGR STATEMENT

You turn on the Apple high-resolution mode with the command HGR. You'll notice that this is the same as the command for entering the low-resolution mode, with the addition of the letter H. This system of transforming low-resolution commands into high-resolution commands with the addition of the letter H is used several times in Applesoft BASIC, as you will see.

Put your computer in TEXT mode, if it is not in that mode already. Then type the HGR command in immediate mode and press RETURN. Just as when you enter low-resolution graphics mode, the screen will turn black, with a text window at the bottom of the display. As before, the text window can be eliminated with a POKE −16302,0. It can be turned back on with a POKE −16301,0.

How do we draw on the high-resolution display? Much as we draw on the low-resolution display, except the command that turns pixels on and off is called HPLOT. (Note, once again, the H to indicate high-resolution graphics.) The HPLOT command works in much the same manner as the PLOT command. We follow the word HPLOT with a pair of numbers, the coordinates of the pixel that we wish to turn on or off. However, we now have 280 possible horizontal positions (from 0 to 279) and 192 possible vertical coordinates (from 0 to 191), to befit the greater resolution



Pinball Construction Set, a game created by Bill Budge and sold by Electronic Arts

(and therefore smaller pixel size) of this new graphics mode.

We determine the color of the pixel plotted with HPLOT by using the HCOLOR command. (There's that H again!) However, we now have considerably fewer colors at our disposal, six to be precise, and a different system of numbers to describe them. Here are the six high-resolution colors:

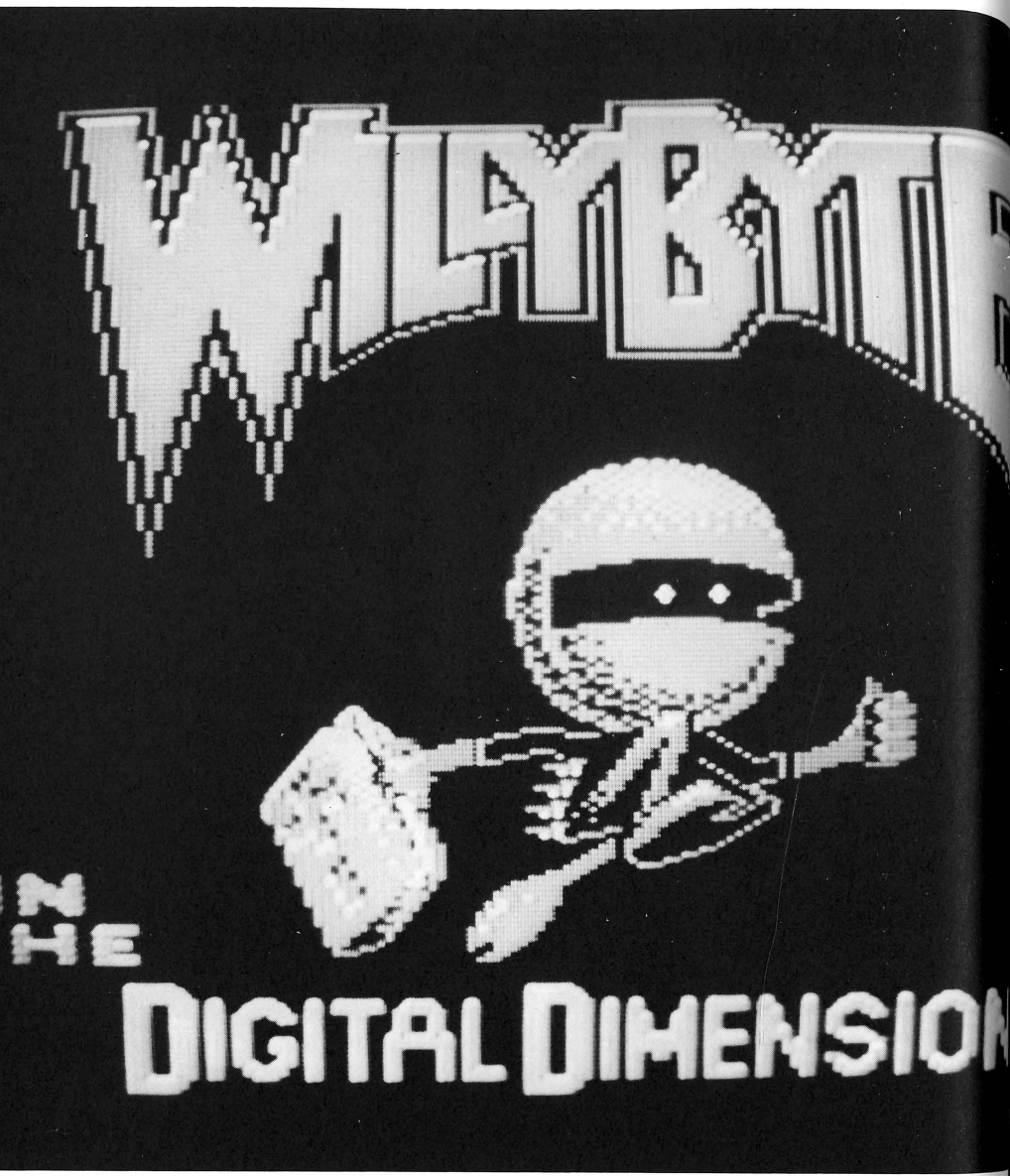
Black1 — 0	Black2 — 4
Green — 1	Orange — 5
Purple — 2	Blue — 6
White1 — 3	White2 — 7

No, the chart hasn't been misprinted. We use eight numbers to describe the six Apple high-resolution colors. Two colors, black and white, are each given two numbers. There is a reason for this, as we shall see momentarily. Although the two blacks and the two whites are differentiated on the chart by the addition of a 1 and a 2 after their names, they will look quite the same when you display them on the high-resolution screen. Nonetheless, they are different.

Confused already? Then take a deep breath and prepare yourself for the worst, because things go downhill from here. A lot of things about the Apple high-resolution mode don't make sense at first glance, and some of them don't make sense at second or third glance either. That doesn't mean that you won't be able to learn how to program in high-resolution mode; it just means that you'll have to sit at your computer and play with high-resolution graphics until things start to add up. You still might not understand *why* they work that way, but at least you'll know *how* they work, and you'll be able to work with them. Creating high-resolution pictures isn't really hard, once you learn to work within the (admittedly odd) system.

PLAYING WITH HIGH RESOLUTION

To help you get your feet wet a toe at a time, try a couple of simple high-resolution commands in the immediate mode. Put your Apple in HGR mode, if it isn't already, and type these two commands:



Willy Byte revisited

```
HCOLOR = 2
HPLOT 10,10
```

pressing RETURN after each. A purple pixel should appear way up in the upper left-hand corner of the screen. It is a very small pixel, and you may have a little trouble finding it, after the very large pixels of the low-resolution mode. After all, purple is a rather dark color and the black background of the display is darker still. Nonetheless, the pixel is there, if you look for it.

Once you've found the purple pixel, type the following commands in the immediate mode:

```
HCOLOR = 5
HPLOT 151,100
```

pressing RETURN after each. An orange pixel will appear somewhere near the middle of the display.

HPLOTting a pixel would seem to be a simple matter indeed, just like PLOTting a pixel in low-resolution mode. However, sometimes appearances can be deceiving. There is more involved here than meets the eye. Type the following commands in the immediate mode:

```
HCOLOR = 6
HPLOT 205,50
```

pressing RETURN after each. If you'll consult the color chart above, you'll see that these commands should have drawn a blue pixel, somewhere in the upper right-hand quarter of the screen. Only, no pixel appeared. What went wrong?

We have just run smack-dab against one of the idiosyncracies of the Apple high-resolution mode. Sometimes, a pixel that you try to display on the high-resolution screen isn't displayed at all. This never happened in the low-resolution mode!

Now type these commands in the immediate mode:

```
HCOLOR = 7
HPLOT 254,95
```

pressing RETURN after each. According to the color chart, this should HPLOT a white pixel toward the right-hand

side of the display. And, sure enough, a pixel appears when we type these commands—but it isn't white, it's blue!

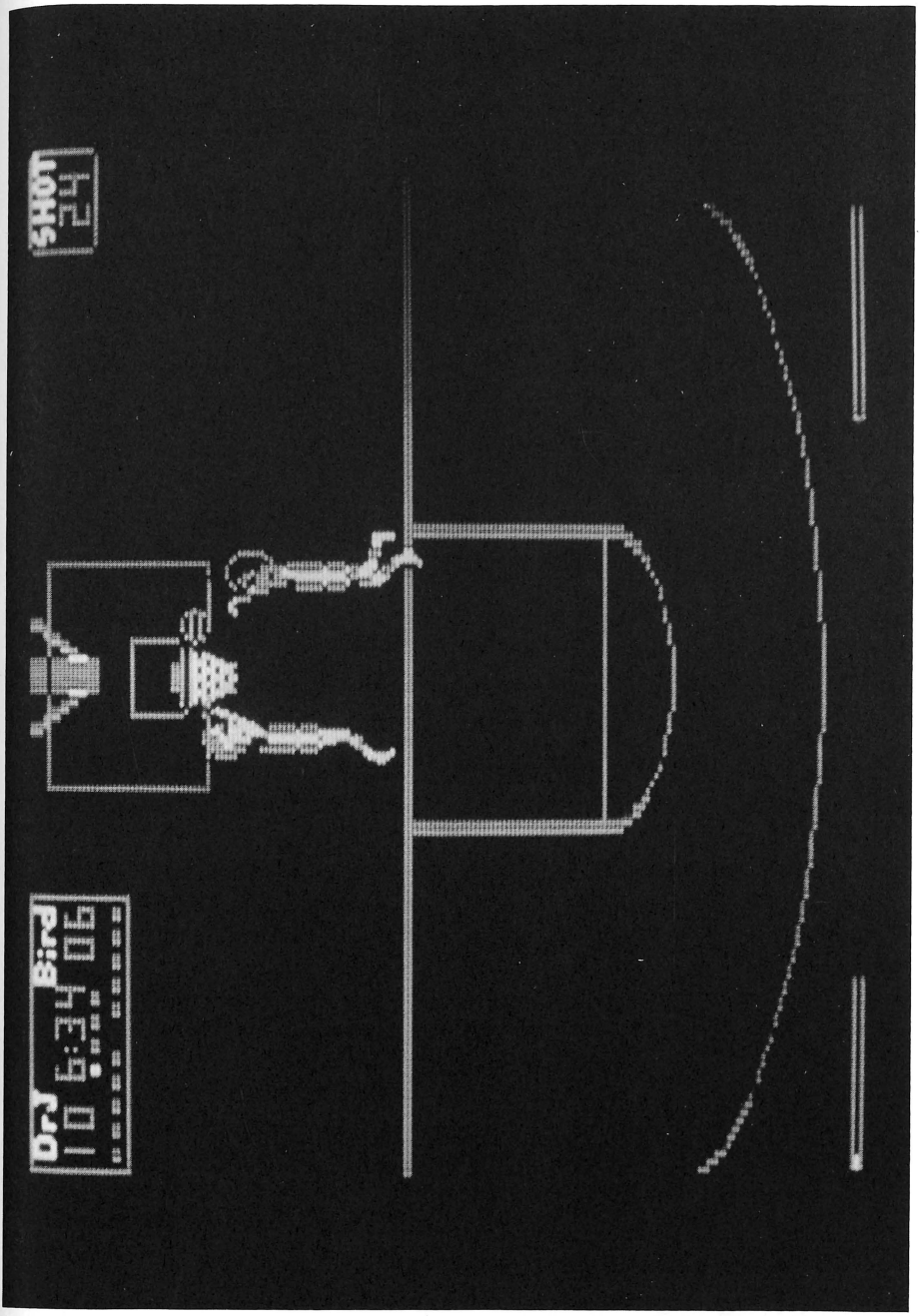
Sort of makes you want to go back to watercolors, doesn't it? But, take heart! While the high-resolution mode may have its idiosyncracies, they are at least consistent idiosyncracies. There is a method behind all of this madness. A complicated method, but a method nonetheless. If it helps, you might want to make a game out of figuring out what this method is. Play around a little in the immediate mode. HPLOT pixels at various coordinates and in various colors. See if you can figure out why the pixels sometimes appear in the right colors and sometimes appear in the wrong colors and sometimes don't appear at all. And, if you can't figure out why (which would really require some knowledge of computer hardware, a subject we haven't gone into in this book), at least see if you can find a consistent pattern to the way in which the pixels do or do not appear.

Then, when you finish exploring, come back to this book and you'll learn some more about how Apple high-resolution graphics work.

THE METHOD IN THE MADNESS

First, let's consider the colors green, purple, orange, and blue. If you'll consult the chart earlier in this chapter, you'll see that two of these colors (green and orange) have been assigned odd-numbered codes (1 and 5) and that two of these colors (purple and blue) have been assigned even-numbered codes (2 and 6).

There is a significance to this numbering. The odd-numbered colors, green and orange, can be displayed only at odd-numbered horizontal coordinates; and the even-numbered colors, purple and blue, can be displayed only at even-numbered coordinates. If you'll glance back several paragraphs, you'll see that this explains why we were unable to display a blue pixel at a horizontal coordinate of 205. Blue is an even-numbered color, with a code of 6, but we attempted to display it at an odd-numbered coordinate position. Because such a thing cannot be done, the Applesoft interpreter refused to do it. No pixel was displayed. On the other hand, if we had tried to display it at horizontal coordinate 204 or horizontal coordinate 206, we would have had no problem.



One-on-One, a game distributed by Electronic Arts

Black and white, on the other hand, present something of a special problem. Black pixels can be displayed at both even- and odd-numbered horizontal coordinates. White pixels, on the other hand, can't be displayed in either odd- or even-numbered horizontal coordinates; white pixels will appear only when two nonblack pixels are displayed at adjacent horizontal coordinates. Although this may sound rather confusing, it should become clear after a quick demonstration. Type the following commands in the immediate mode:

```
HCOLOR = 6
HPLOT 100,10
HCOLOR = 5
HPLOT 101,10
```

Although common sense (and the color chart) tells us that this should display a blue pixel next to an orange pixel, this is not the case. Actually, it displays a pair of white pixels, because two nonblack pixels at adjacent horizontal coordinates always become white pixels. Of course, we could also display a pair of white pixels like this:

```
HCOLOR = 7
HPLOT 100,10
HPLOT 101,10
```

And this, of course, is a more sensible method of displaying a pair of white pixels, because it uses fewer commands and because it makes a lot more sense to us as programmers. COLOR 7, after all, is the code for white.

If we attempt to display a single white pixel, without a nonblack pixel at an adjacent horizontal coordinate, the white pixel will in fact display in another color. For instance, if we HPLOT a lone white pixel at an odd-numbered coordinate, it will appear as either green or orange. If we HPLOT it at an even-numbered coordinate, it will appear as either purple or blue. To see this process in action, type the following commands in the immediate mode:

```
HCOLOR = 3
HPLOT 121,130
```

This should display a green pixel in the lower middle of the screen.

A QUICK REVIEW

If you find all of this rather baffling, you will be less than thrilled to hear that still more idiosyncracies of the Apple high-resolution mode will be explained in a moment. Until then, it is important that you have a solid grasp of the idiosyncracies just discussed. So we will review them here:

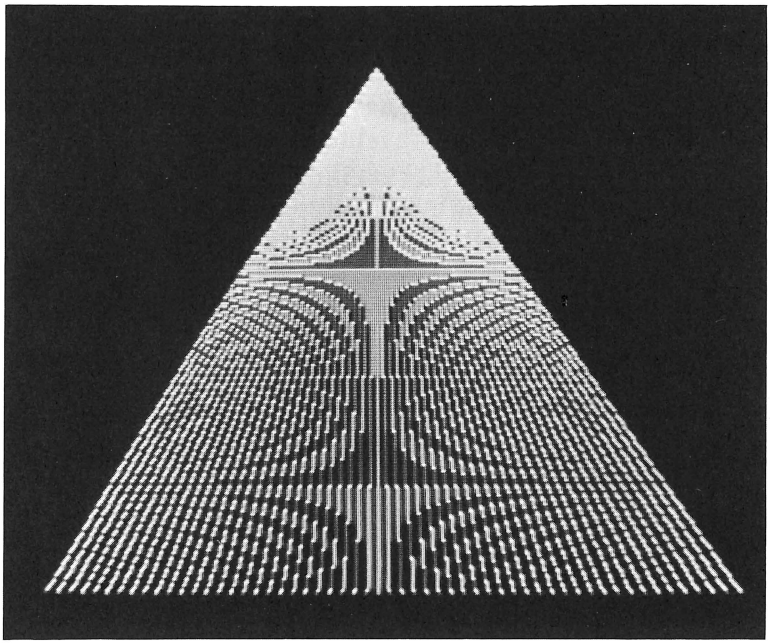
- Black pixels may be displayed at any horizontal coordinate position.
- Green and orange pixels may be displayed only at odd-numbered horizontal coordinates.
- Purple and blue pixels may be displayed only at even-numbered horizontal coordinates.
- A lone white pixel at an odd-numbered horizontal coordinate will appear green or orange.
- A lone white pixel at an even-numbered horizontal coordinate will appear purple or blue.
- Two nonblack pixels at adjacent horizontal coordinates will appear white.

If any of the above statements is less than clear, experiment in the immediate mode until you see the sense of it. Even if you don't understand why the Apple high-resolution mode works the way it does, be sure that you understand just how it works.

And, after you've finished experimenting, type an HGR command in the immediate mode. This will clear all of the old pixels off the high-resolution display, preparing it for still more experimentation yet to come.

COLOR GROUPS

The other idiosyncracy of the Apple high-resolution mode has to do with our ability to display certain colors in close juxtaposition to one another, that is, within a range of seven or fewer horizontal coordinates.



Original design by Geyer, Geyer, and Halpern

It is easy enough to display a green pixel in close proximity to a purple pixel. Type the following commands in the immediate mode:

```
HCOLOR = 2
HPLOT 0,0
HCOLOR = 1
HPLOT 3,0
```

pressing RETURN after each. This puts a purple pixel and a green pixel practically next to one another in the upper left-hand corner of the display. (If we put them any closer, they would turn into a pair of white pixels.) Similarly, it is easy to display an orange pixel in close proximity to a blue pixel.

Type the following commands in the immediate mode:

```
HCOLOR = 5
HPLOT 273,0
HCOLOR = 6
HPLOT 276,0
```

pressing RETURN after each. This puts an orange pixel very close to a blue pixel, in the upper right-hand corner of the display.

It is not so easy, however, to put a green pixel near an orange pixel, or a purple pixel near a blue pixel. Type the following commands in the immediate mode:

```
HCOLOR = 1
HPLOT 1,100
```

pressing RETURN after each. This put a green pixel on the left-hand side of the screen, about halfway down the display. Now type the following commands in the immediate mode:

```
HCOLOR = 5
HPLOT 3,100
```

pressing RETURN after each. This puts an orange pixel next to the green pixel, only . . . as soon as the orange pixel appears, the green pixel turns orange!

Well, nobody ever claimed that high-resolution graphics were simple—or sensible! The fact is, we can't go putting orange and green pixels that close together on the high-resolution screen without careful planning. The same goes for purple and blue pixels, green and blue pixels, and purple and orange pixels.

Take one more look at the high-resolution color chart, if you haven't already memorized it. Notice that it is arranged in two columns of four colors each. This columnar arrangement has a special significance. We will refer to the colors in the first column (black1, green, purple, white1) as *group 1 colors.* We will refer to the colors in the second column (black2, orange, blue, white2) as *group 2 colors.*

Group 1 colors are not easily mixed with group 2 colors. They may be displayed together on the same screen, and

even on the same horizontal line, but they cannot always be displayed close together.

LINE SEGMENTS

Each line of the high-resolution screen consists of 280 pixels. Imagine that this line is broken up into forty segments of 7 pixels each. The first such segment extends from horizontal coordinate 0 to horizontal coordinate 6. The second such segment extends from horizontal coordinate 7 to horizontal coordinate 13. The third such segment extends from horizontal coordinate 14 to horizontal coordinate 20. And so forth, up to the fortieth segment, which extends from horizontal coordinate 273 to horizontal coordinate 279.

Within each seven-pixel segment, all pixels must be in either group 1 colors or group 2 colors. The two groups cannot be mixed within a single segment. If you attempt to plot a pixel in a color from one group in a segment that already contains pixels plotted in colors from the other group, all pixels within the segment will immediately be converted to the equivalent colors in the other group (green to orange and purple to blue, or vice versa).

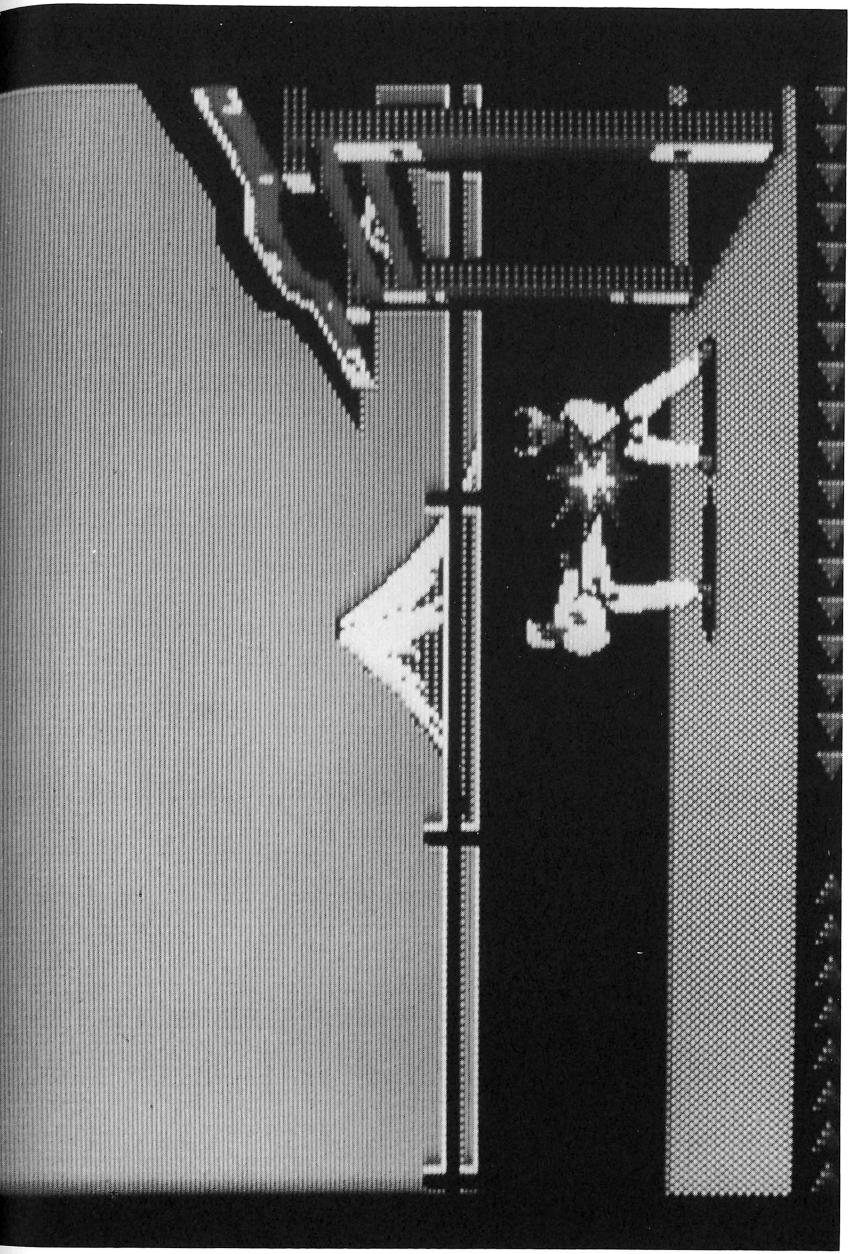
This is why we said earlier that black1 and black2 (or white1 and white2) were different colors, despite the fact that they look the same on the display. If, for instance, you attempt to plot black1 (HCOLOR 0) in a segment that contains group 2 colors, you will convert them to group 1 colors. So if you want to plot a black pixel in a group 2 segment, you must use black2 (HCOLOR 4), even though there is no obvious visual difference between the colors. The same is true for white1 and white2.

If you quiver at the thought of calculating what segment you are plotting pixels in, to avoid accidentally mixing color groups within a segment, there are two possible approaches you can take:

1. Use only colors from one color group when you create a picture, or

2. Keep colors from the two color groups in widely separated areas of the screen.

On the other hand, if it is important to you to make full use of all six high-resolution colors, you must resign yourself to carefully calculating screen segments to avoid mixing colors unfortuitously.



Karateka is a Brøderbund computer game.

And, if you are using a monochrome display (as many Apple users do), you may be wondering what all the fuss is about. All of these pixels (except for the black ones) look alike to you. Bear in mind, however, that you may be called on at some time to write programs that will be used by someone with a color display, and *then* you'll wish that you had paid closer attention during this chapter!

By now, you should have a pretty good notion of how Apple high-resolution graphics work. In the next chapter, we'll begin to put those high-resolution graphics to work, with the aid of the surprisingly versatile HPLOT command.

Suggested Projects

1. Write a program that puts pixels of four different colors on the first line of the screen, between horizontal coordinates 0 and 6.

2. Write a program that draws a horizontal line on the display, using high-resolution pixels. (See the next chapter for an easier method of writing such a program.)

3. Write a program that draws a vertical green line from coordinates 2,0 to coordinates 2,100, then (after a brief delay) draws a vertical blue line from coordinates 3,0 to coordinates 3,100. What happens to the green line when the blue line is drawn? Why?

6
HIGH-RESOLUTION LINES

Drawing a line on the Apple high-resolution display isn't difficult. We can, if we wish, use the same line-drawing technique that we used in the last chapter. Type and RUN this program:

```
10 HGR : HOME
20 HCOLOR = 1
30 FOR X = 1 TO 279 STEP 2
40 HPLOT X,100
50 NEXT X
60 GET A$
70 TEXT
```

When you first turn on high-resolution mode, you will notice a brief glimpse of earlier drawings if you have run other BASIC high-resolution programs since the computer was turned on. Ignore these images; they are a result of BASIC's tardiness in clearing the high-resolution screen.

The above program draws a horizontal line from coordinates 1,100 to 279,100, bisecting the display. Notice that we only plot every other coordinate in the line—specifically, the *odd* horizontal coordinates—so that the line will be in high-resolution color #1 (green) rather than white.

However, there is an easier way to draw a high-resolution line on the Apple. Type and RUN this program:

```
10 HGR : HOME
20 HCOLOR = 1
30 HPLOT 1,100 TO 279,100
40 GET A$
50 TEXT
```

The FOR-NEXT loop in the earlier program has now been reduced to a single HPLOT command. How did we do this? The Applesoft HPLOT command is a lot more than just a high-resolution version of the PLOT command. It does more than just draw pixels; it can also draw lines. (No, there aren't any Applesoft commands called HVLIN or HHLIN.)

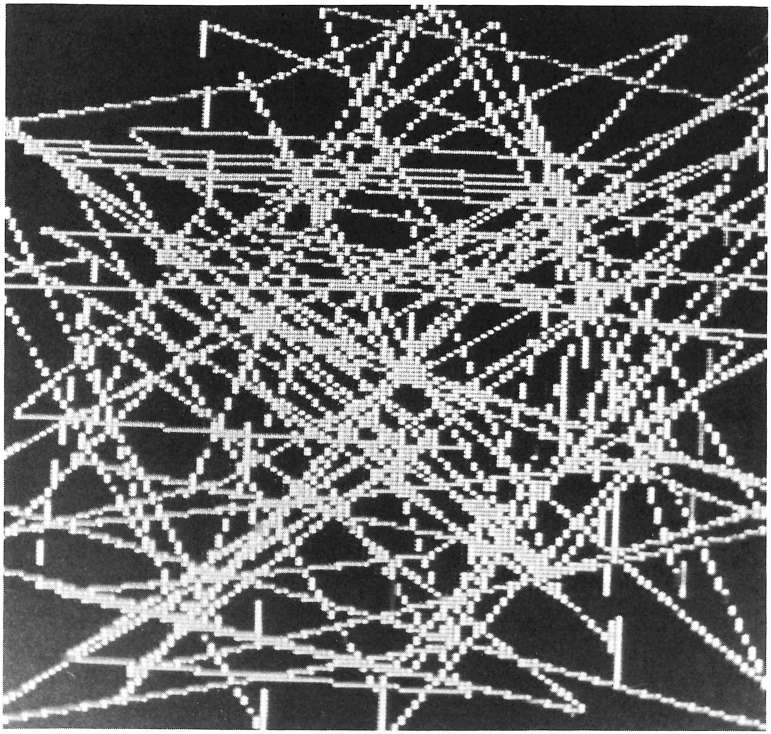
We use the line-drawing version of HPLOT like this:

```
HPLOT x1,y1 TO x2,y2
```

where x1 and y1 are the horizontal and vertical coordinates of the first point in the line and x2 and y2 are the horizontal and vertical coordinates of the last point in the line.

Thus, we can draw a horizontal line as illustrated in the program above. Similarly, we can draw a vertical line by appropriately adjusting the endpoints of the line. In this way, the HPLOT command can be made to imitate both the HLIN and VLIN commands, but with high-resolution lines. However, the HPLOT command can do the HLIN and VLIN commands one better: it can also draw diagonal lines, slanted at any desired angle. In fact, it can draw lines between any two specified points on the display. This program will draw random lines across the display until you press a key:

```
10 HGR
20 POKE -16302,0
30 HCOLOR = 5
40 X1 = INT(RND(1) * 280) : Y1 = INT(RND(1) * 192)
50 X2 = INT(RND(1) * 280) : Y2 = INT(RND(1) * 192)
60 HPLOT X1,Y1 TO X2,Y2
70 X1 = X2 : Y1 = Y2
80 IF PEEK (-16384) < 128 THEN 50
90 TEXT
```

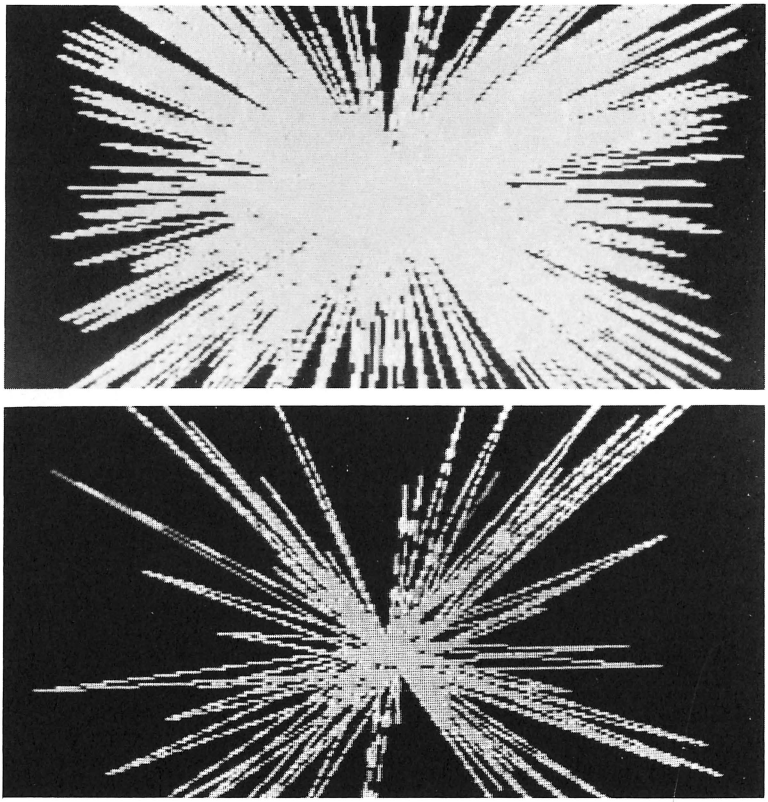


Random lines created using author's program

Lines 40 and 50 generate random endpoints for the line to be drawn and line 60 draws it. Notice, however, that line 40 executes only once, to calculate the starting point of the first line. After that, line 70 passes the value of X2 and Y2 to X1 and Y1, so that the ending point of one line becomes the beginning point of the next line. Line 80 PEEKs at memory to see if a key is being pressed. If not, it loops back to draw the next line; if so, the program ends and returns to text mode.

LOTS OF LINES

The HPLOT command can draw more than one line at a time, using this syntax:



Geyer, Geyer, and Halpern modified the author's random-line program to produce these designs.

```
HPLOT x1,y1 TO x2,y2 TO x3,y3  . . .
```

for as many consecutive points as you can cram into a program line. Each of these points will be connected by a continuous series of lines. In this way, the HPLOT command can be used to draw shapes on the screen. For instance, the following program draws a rectangle:

```
10 HGR
20 POKE -16302,0
30 HCOLOR = 2
40 HPLOT 50,20 TO 150,20 TO 150,120 TO 50,120 TO 50,20
```

```
50 GET A$
60 TEXT
```

Obviously, the HPLOT command is quite versatile. However, the fact that it will simply duplicate the work of the HLIN and VLIN commands opens up a number of possibilities. For instance, we can use HPLOT to create filled rectangles on the high-resolution display. Type this program and RUN it:

```
10 HGR
20 HCOLOR = 6
30 POKE -16302,0
40 FOR X = 50 TO 200
50 HPLOT X,50 TO X,150
60 NEXT X
70 GET A$
80 TEXT
```

The result is a big, fat blue rectangle in the middle of the display.

What good does this do us? Well, among other things, it means we can adapt our bar chart program to high-resolution mode. Because we can cram far more high-resolution pixels than low-resolution pixels into a display, this will give us the ability to create a larger number of bars, and to display finer differences between the size of two bars. To adapt the bar chart program to high-resolution mode, make the following changes to the original program:

```
10 SW = 280 : SH = 160 : MB = 30
160 HGR : HOME
170 HCOLOR = 3
180 W=SW/LB
240 HPLOT I * W,Y TO I * W + W - 1, Y
```

These may seem like minor changes, but they will turn the bar chart program into a full-fledged, high-resolution graphics program, and the results will be impressive. You will now be able to create a full thirty bars on the display, and very small differences between bars will now be visible, which they were not in the low-resolution version. Once you've made these changes, RUN the new version of the program and play with it.

After creating a few charts, you'll notice that the new version is slower than the old one. That's because we now have so many more pixels to draw in each bar. However, this is a small price to pay for the extra power.

With our newfound ability to draw high-resolution lines, we can create a new kind of chart as well: a line graph. Like a bar chart, the graph shows the relation between a series of numbers, by representing each number as a point and connecting those points with a continuous line, running from left to right on the display. The following program creates such a graph much as the last program created a bar chart. First it will ask you for the number to be represented by each point on the chart, then it will draw the chart. Feel free to add any embellishments to the program that might seem worthwhile.

```
10 DIM P(100)
20 HP = 0
30 TEXT : HOME
40 INPUT "NAME OF CHART? "; NC$
50 INPUT "NUMBER OF POINTS? "; NP
60 IF NP > 100 THEN PRINT "TOO LARGE. TRY AGAIN." :
GOTO 50
70 IF NP < 2 THEN PRINT "TOO SMALL! TRY AGAIN." :
GOTO 50
80 FOR I = 1 TO NP
90 PRINT "VALUE FOR POINT "; I; : INPUT P(I)
100 IF P(I) < 1 THEN PRINT "BAD VALUE! TRY AGAIN." :
GOTO 90
110 IF HP < P(I) THEN HP = P(I)
120 NEXT
130 HOME
140 HGR
150 HCOLOR = 3
160 HPLOT 0,0 TO 0,159 TO 279,159
170 VTAB 22 : HTAB 20 - LEN (NC$) / 2
180 PRINT NC$
190 SU = 160/HP : W = 279/(NP - 1)
200 FOR I = 1 TO NP - 1
210 HPLOT (I - 1) * W, 160 - SU * P(I) TO I * W, 160 - SU * P(I
+ 1)
220 NEXT
230 GET A$
240 GOTO 20
```

FANCY SHAPES

Because the HPLOT command does not restrict us to horizontal and vertical lines, we can draw a much greater variety of shapes. Here is a short program that turns our rectangle-drawing program sideways and draws a filled diamond on the display:

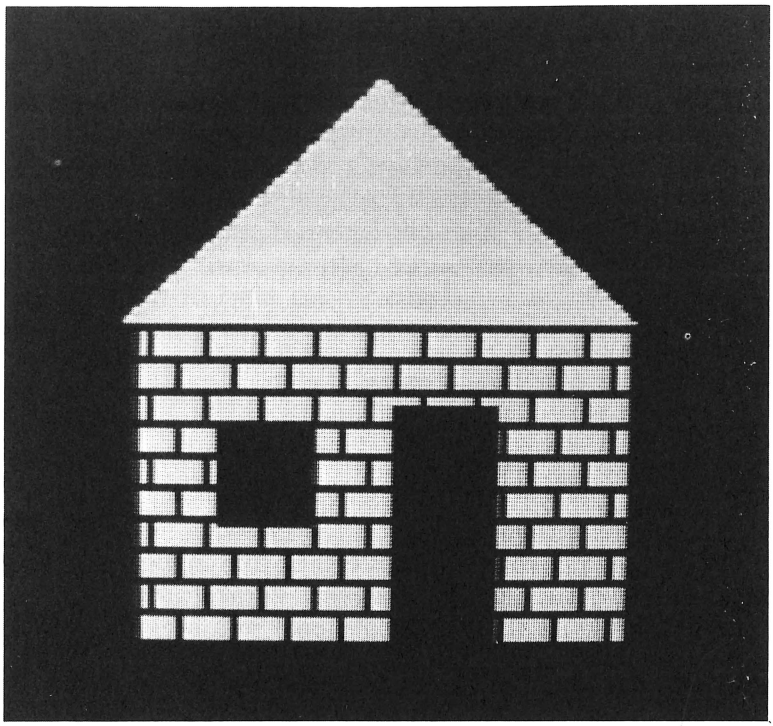
```
10 HGR
20 HCOLOR = 5
30 POKE -16302,0
40 FOR I = 0 TO 30
50 HPLOT 50 + I, 85 + I TO 80 + I, 20 + I
60 NEXT I
70 GET A$
80 TEXT : HOME
```

Complex networks of lines can be drawn with HPLOT, and thus we can create relatively complicated pictures. Here's a program that draws a sketch of a house:

```
10 HGR
20 HCOLOR = 7
30 POKE -16302,0
40 HPLOT 70,80 TO 210,80 TO 140,20 TO 70,80 TO 70,160
TO 210,160 TO 210,80
50 HPLOT 92,130 TO 92,105 TO 120,105 TO 120,130 TO 92,130
60 HPLOT 144,160 TO 144,100 TO 170,100 TO 170,160
70 GET A$
80 TEXT : HOME
```

Although this picture of a house is more complicated than the shapes we have drawn so far, it is still rather unsophisticated and lacking in detail. Of course, we can add greater and greater amounts of detail by simply increasing the number of HPLOT statements and the number of lines. But such a program would become increasingly cumbersome.

One way to get around this is to write subroutines using HPLOT statements that will draw simple shapes that can in turn be used as building blocks to draw more complex shapes. For instance, the following program contains a subroutine at line 500 that draws a simple pattern of bricks at the horizontal and vertical coordinates contained in vari-



The house that Lampton built

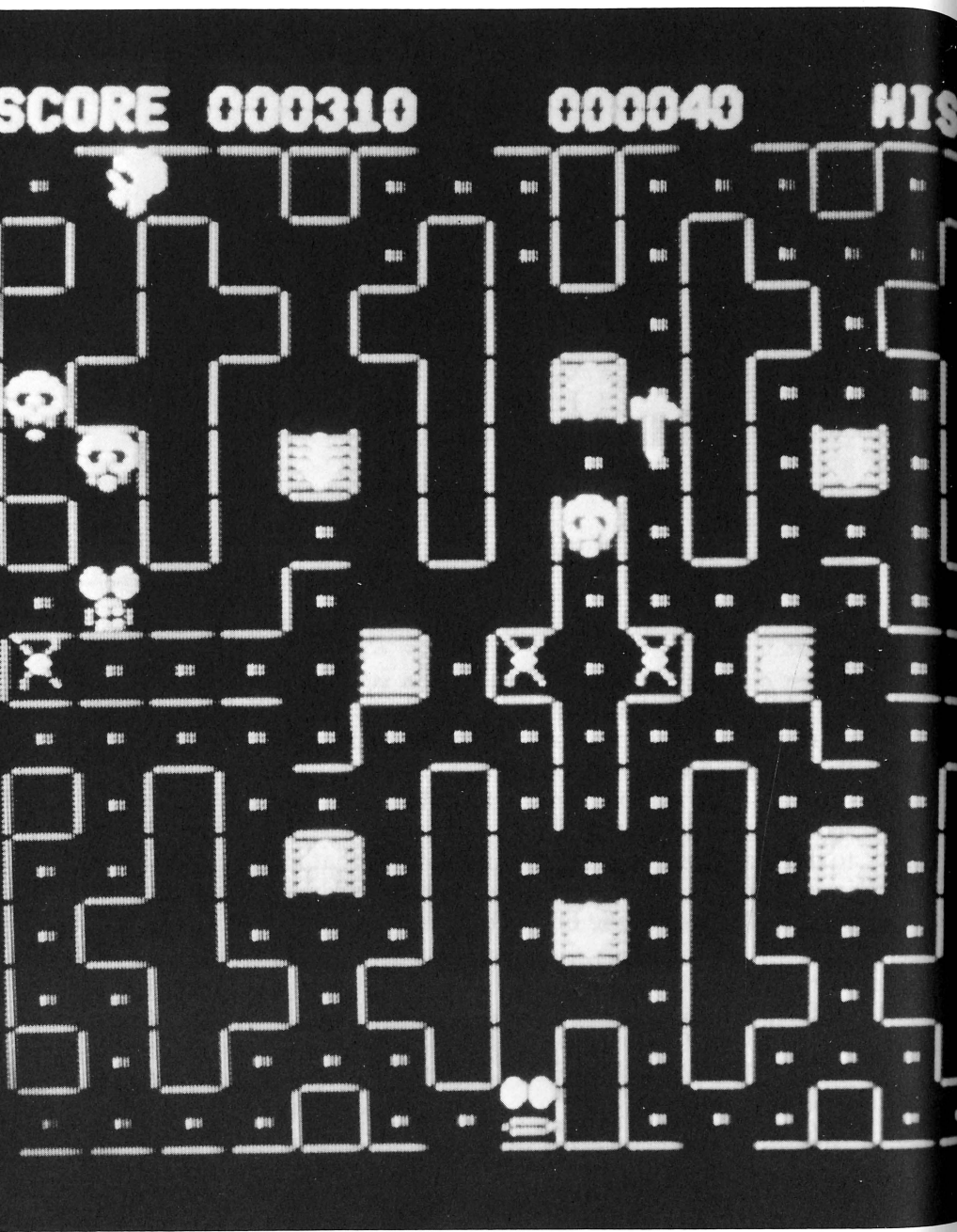
ables X and Y. The main program calls this subroutine repeatedly, to build up the image of a house, not unlike the one sketched above, built out of bricks:

```
10 HGR
20 POKE -16302,0
30 FOR X = 70 TO 210 STEP 16 : REM Draw body of house.
40 FOR Y = 80 TO 159 STEP 16
50 GOSUB 500
70 NEXT Y : NEXT X
80 HCOLOR = 3
90 FOR Y = 79 TO 20 STEP -1 : REM Draw pointed roof.
100 HPLOT 65 + 1.25 * (79 - Y), Y TO 215 - 1.25 * (79 - Y), Y
110 NEXT Y
120 HCOLOR = 0
130 FOR I = 0 TO 25 : REM Draw window.
```

```
140 HPLOT 92, 105 + I TO 120, 105 +I
150 NEXT I
160 FOR I = 0 TO 60 : REM Draw door.
170 HPLOT 144,100 + I TO 174, 100 + I
180 NEXT I
190 GET A$ : REM Wait for keypress.
200 TEXT
210 END
499 REM The following subroutine draws a segment of a brick
wall at the coordinates contained in variables X and Y.
500 HCOLOR = 3
510 FOR I = 2 TO 15
520 HPLOT X, Y + I TO X + 15, Y + I
530 NEXT I
540 HCOLOR = 0
550 HPLOT X, Y + 8 TO X + 15, Y + 8
560 HPLOT X, Y + 9 TO X + 15, Y + 9
570 HPLOT X + 2, Y + 2 TO X + 2, Y + 7
580 HPLOT X + 3, Y + 2 TO X + 3, Y + 7
590 HPLOT X + 10, Y + 8 TO X + 10, Y + 15
600 HPLOT X + 11, Y + 8 TO X + 11, Y + 15
610 RETURN
```

Now that we have a subroutine that can draw segments of a brick wall, we can use that subroutine to draw anything that might conceivably be made of bricks. The following program uses the brick-drawing subroutine (at 500) plus a second subroutine that draws segments of a ladder (at 700) to draw a series of platforms and ladders that might be used in a "running, jumping, and climbing" style arcade game, such as *Donkey Kong* or *Lode Runner*:

```
10 HGR : HOME
20 POKE -16302,0
30 GOSUB 1000 : REM DRAW PLATFORMS.
40 GET A$ : REM WAIT FOR KEYPRESS.
50 TEXT
60 END
499 REM The following subroutine draws a segment of a brick
wall at the coordinates contained in variables X and Y.
500 HCOLOR = 3
510 FOR I = 2 TO 15
520 HPLOT X, Y + I TO X + 15, Y + I
```



Lode Runner, from Brøderbund

```
530 NEXT I
540 HCOLOR = 0
550 HPLOT X, Y + 8 TO X + 15, Y + 8
560 HPLOT X, Y + 9 TO X + 15, Y + 9
570 HPLOT X + 2, Y + 2 TO X + 2, Y + 7
580 HPLOT X + 3, Y + 2 TO X + 3, Y + 7
590 HPLOT X + 10, Y + 8 TO X + 10, Y + 15
600 HPLOT X + 11, Y + 8 TO X + 11, Y + 15
610 RETURN
699 REM The following subroutine draws a segment of a ladder
at the coordinates contained in variables X and Y.
700 HCOLOR = 0
710 FOR I = 0 TO 15
720 HPLOT X, Y + I TO X + 15, Y + I
730 NEXT I
740 HCOLOR = 3
750 HPLOT X + 2, Y TO X + 2, Y + 15
760 HPLOT X + 3, Y TO X + 3, Y + 15
770 HPLOT X + 10, Y TO X + 10, Y + 15
780 HPLOT X + 11, Y TO X + 11, Y + 15
790 HPLOT X + 4, Y + 3 TO X + 9, Y + 4
800 HPLOT X + 4, Y + 4 TO X + 9, Y + 4
810 HPLOT X + 4, Y + 11 TO X + 9, Y + 11
820 HPLOT X + 4, Y + 12 TO X + 9, Y + 12
830 RETURN
999 REM The following subroutine draws a sequence of
platforms and ladders using information stored in DATA
statements.
1000 READ X1 : REM Get horizontal coordinate of platform.
1010 IF X1 < 0 THEN 1080 : REM If last platform, skip ahead.
1020 READ Y1, H , W : REM Read vertical coordinate, height,
width.
1030 FOR X = X1 to X1 + (W - 1) * 16 STEP 16 : REM Draw
platform.
1040 FOR Y = Y1 TO Y1 + (H - 1) * 16 STEP 16
1050 GOSUB 500 : REM Draw brick segment.
1060 NEXT Y : NEXT X
1070 GOTO 1000 : REM Draw next platform, if any.
1080 READ X : REM Read horizontal coordinate of ladder.
1090 If X < 0 THEN RETURN : REM If last ladder, return.
1100 READ Y1,H : REM Read vertical coordinate, height.
1110 FOR Y = Y1 TO Y1 + (H - 1) * 16 STEP 16 : REM Draw
ladder.
```

```
1120 GOSUB 700 : REM Draw ladder segment.
1130 NEXT Y
1140 GOTO 1080 : REM Draw next ladder, if any.
1999 REM Data for platforms: horizontal and vertical coordinates,
height and width.
2000 DATA 20, 150, 1, 9 : REM Platform #1
2010 DATA 68, 88, 1, 12 : REM Platform #2
2020 DATA 179, 56, 1,6 : REM Platform #3
2030 DATA 40, 26, 1, 12 : REM Platform #4
2040 DATA 200, 120, 3, 5 : REM Platform #5
2050 DATA -1 : REM End of platforms.
2099 REM Data for ladders: horizontal and vertical coordinates,
and height.
2100 DATA 100, 88, 4 : REM Ladder #1
2110 DATA 200, 26, 4 : REM Ladder #2
2120 DATA 232, 88, 6 : REM Ladder #3
2130 DATA -1 : REM End of ladders.
```

Type this program and RUN it. The key to the program is in the subroutine at 1000 and the DATA statements at 2000. This final subroutine draws the ladders and platforms by repeatedly calling the brick- and ladder-drawing routines at 500 and 700, respectively. However, the pattern in which the platforms and ladders are drawn is contained in the DATA statements. Each platform is described by four numbers: the horizontal coordinate of the platform, the vertical coordinate of the platform, the height of the platform, and the width of the platform. As many platforms as desired can be described in the DATA statements. The end of the platform descriptions is marked by the number −1 (or any other negative number). This is then followed by the descriptions of the ladders, which are similar but not identical to the descriptions of the platforms. Each ladder description contains three numbers: the horizontal coordinate of the ladder, the vertical coordinate of the ladder, and the height of the ladder. All ladders have the same width, so no additional description is necessary. The end of the ladder descriptions is marked with a −1, or other negative number.

By changing the DATA statements, you can change the pattern of platforms and ladders on the display. Try it and see, following the rules given in the previous paragraph for creating platform and ladder descriptions.

Furthermore, if these subroutines were to be used to create an arcade game, they could easily be used to draw a *series* of screens with different patterns of bricks and ladders on each, simply by adding additional DATA statements containing additional platform and ladder descriptions, and calling the subroutine at line 1000 each time a new screen was to be drawn. For instance, if you make the following additions and changes to the program above, it will clear the screen and draw a second screen of platforms and ladders as soon as you press a key:

```
45 IF X <> - 2 THEN 10
2200 DATA 100, 154, 1, 5
2210 DATA 52, 122, 1, 5
2220 DATA 148, 122, 1, 5
2230 DATA 4, 90, 1, 5
2240 DATA 196, 90, 1, 5
2250 DATA 52, 58, 1, 5
2260 DATA 148, 58, 1, 5
2270 DATA 100, 26, 1, 5
2280 DATA -1
2300 DATA 116, 122, 2
2310 DATA 148, 122, 2
2320 DATA 68, 90, 2
2330 DATA 196, 90, 2
2340 DATA 52, 58, 2
2350 DATA 212, 58, 2
2360 DATA 100, 26, 2
2370 DATA 164, 26, 2
2380 DATA -2
```

You can add your own additional screens to this program, by changing the −2 in the DATA statement in line 2380 and adding the new screen descriptions in succeeding DATA statements. The routine from line 10 to line 45 will continue drawing new screens until it finds one in which the ladder descriptions are terminated by a −2 rather than a −1; thus, terminate your final screen with a −2.

SAVING THE ENDPOINTS

The last several programs demonstrate how complex pictures can be built up out of simple building blocks, a com-

mon high-resolution graphics technique. Often, we will store the information necessary for creating these complex pictures in DATA statements, as we did above. Another way in which we can use DATA statements is to store the endpoints of lines in a complicated shape. We can then draw the shape by reading the endpoints from the DATA statements and drawing lines between them with a simple HPLOT statement placed in a program loop.

Here, for example, is a program that draws the same house as our earlier house-drawing program but does so using information stored in DATA statements:

```
10 HGR
20 POKE -16302,0
30 COLOR=3
40 READ X1,Y1 : REM Get first coordinates of drawing.
50 READ X2 : REM Get horizontal coordinate of endpoint.
60 IF X2=-999 THEN 110 : REM If -999 then stop drawing.
70 READ Y2 : REM Get vertical coordinate of endpoint.
80 IF X2 > 0 THEN HPLOT X1,Y1 TO X2,Y2 : REM If X2 is not
negative, draw line.
90 X1 = ABS(X2) : Y1 = Y2 : REM Start next line at end of old.
100 GOTO 50 : REM Draw next line.
110 GET A$ : REM Wait for keypress.
120 TEXT
130 END
499 REM The following lines contain the DATA necessary for
drawing a house.
500 DATA 70, 80, 210, 80, 140, 20, 70, 80, 70, 160, 210, 160,
210, 80
510 DATA -92, 130, 92, 105, 120, 105, 120, 130, 92, 130
520 DATA -144, 160, 144, 100, 170, 100, 170, 160
530 DATA -999
```

The workings of this program are worth discussion. The coordinates for the successive endpoints of the lines making up the house are stored in the DATA statements at the end of the program. The routine on lines 40 to 100 reads these DATA statements and draws the house, connecting each set of coordinates with a white line. In a few cases, there are endpoints that we do not *wish* to have connected to the remainder of the drawing with a white line. To indicate this to the drawing routine, the horizontal coordinate

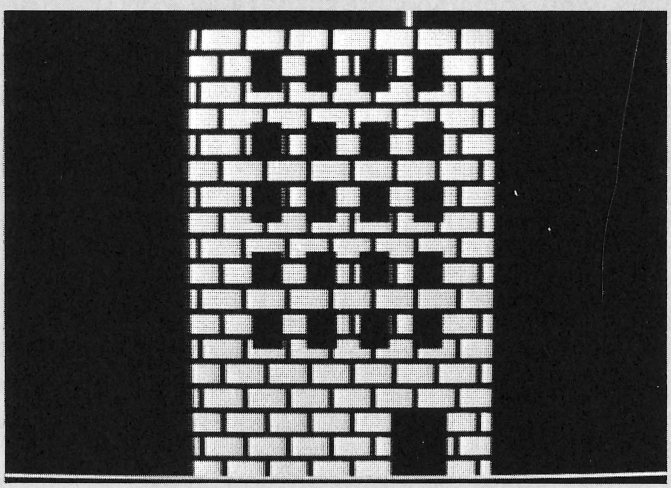
of such an endpoint is made negative. When the drawing routine encounters a negative endpoint, it simply moves the current drawing "cursor" (i.e., the point from which the next line will be drawn) to those coordinates, without drawing a line. When a horizontal coordinate of −999 is encountered, the drawing ends.

You might ask why this kind of drawing program would be preferable to the earlier house-drawing program, which was actually shorter than this program. The answer is that we can extend the complexity of this drawing almost infinitely, simply by adding additional coordinates to the DATA statements. Attempting to do this in the earlier program would have quickly produced an awkward tangle of HPLOT statements. Furthermore, we can easily alter this drawing by changing the DATA statements. In fact, we can plug in an entirely new drawing simply by replacing the DATA statements with new ones. And, as in our platform-and-ladders program, we can create several drawings in succession, by adding additional DATA statements and reexecuting the drawing routine until a prearranged number is read for the horizontal coordinate, perhaps −9999.

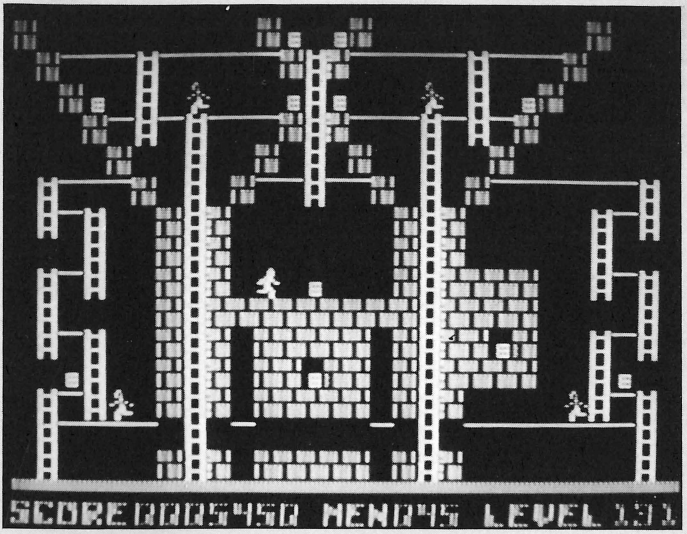
This is a convenient way to draw shapes on the Apple high-resolution display. However, Apple also provides another, similar method: the shape table. This is the subject of our next chapter.

Suggested Projects

1. Rewrite the final version of the house-drawing program to draw different pictures, by changing the DATA statements. Design appropriate DATA statements to draw a picture of a face, a large building, a spaceship, and a flower.



Geyer, Geyer, and Halpern's solution to Project 1



Maze Craze Construction Set (designer: Eric Hammond; graphic artist: Greg Hammond; manufacturer: Data Trek) is a program that allows you to build your own mazes and draw your own animated characters and monsters.

2. Write a program that will draw a maze. The pattern of the maze can be based on a fixed design, or it can be designed randomly by the computer.

7
HIGH-RESOLUTION SHAPES

The shapes that we've drawn so far on the Apple high-resolution display all have one thing in common: they were drawn slowly. We were forced to sit back and watch while the computer drew the various lines that made up the picture. While for some purposes this is quite adequate, it will also prove to be an irritation when we need something drawn *fast*. For instance, it is unlikely that we would ever be able to animate a picture drawn in this fashion.

Fortunately, Applesoft BASIC offers an alternative method of creating shapes that is much faster, more efficient, and versatile: shape tables. There is a catch, however: shape tables are not particularly easy to use, or to understand. You will have to put a good bit of work into learning to use this valuable graphics tool, though the results will be well worthwhile if you are serious about creating graphics on an Apple II series computer. If at any point you should find the material in this chapter beyond your ability to handle, you might want to skip it for now and come back to it at a later point in your computer education. Even without knowing how to create shape tables, you can write many worthwhile graphics programs using Applesoft BASIC. And utility programs are available commercially that will largely automate the process of creating shape tables, so that you will not have to understand the more obscure complexities

of the process. You might want to investigate such utilities; look for them at stores that sell software and computers, or in the ads in the back of computer magazines.

THINKING IN VECTORS

In order to create a shape table, we must start thinking of line drawings in a new way: as a series of *vectors.* Perhaps the best way to understand the concept of vector is by analogy. Imagine that you are drawing a picture on paper, using a ball-point pen. The picture is made up entirely of straight lines oriented to one another at right angles. Although this may seem a terribly limiting way to draw a picture, it isn't really (though it sometimes can be awkward). Even a circle, which would seem to have nothing of straight lines and right angles in it, can be drawn in this fashion, if the lines are made small enough. (Even so, you might want to avoid creating circles with shape tables, if at all possible.) When you draw a line on the paper, or when you lift your pen to put it on another part of the paper to draw a new line, you must move your hand. The movement of your hand has a direction. That direction is a vector.

Since everything in the picture is being drawn at right angles, you can move your hand in only four possible directions: up, down, right, and left. If the pen is touching the paper during this movement, you will draw a line in one of those directions. If you have lifted the pen above the surface of the paper, you will relocate the pen in that direction without drawing. Thus, eight possible vectors can be used in creating a shape table, and we may assign each of them a number:

0 Move up, without drawing
1 Move right, without drawing
2 Move down, without drawing
3 Move left, without drawing
4 Move up, while drawing
5 Move right, while drawing
6 Move down, while drawing
7 Move left, while drawing

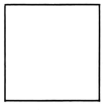
In general, you will want to create a shape on paper (preferably graph paper) before breaking it into vectors.

Even better, you might want to buy one of the graphics utility programs that are available commercially to help you draw such a picture on the video display of your computer, then convert the drawing automatically into a shape table. Barring the availability of such a program, however, you will create your shape table on paper and break it into a series of vectors by hand. Later, you will use the Applesoft BASIC command DRAW to tell the computer to convert the shape table back into a drawing.

If you think ahead a little, you might wonder: When the Apple draws a single vector of my picture, how long a line will it draw? The answer is: As long as you want it to be. Later, when you write a program to draw your painstakingly designed shape on the display, you must issue a SCALE command before you can use the DRAW command. The SCALE command tells Applesoft how long each vector will be, from one pixel to as many pixels as you can squeeze onto the display. All you have to remember in creating your drawing is that each vector must be the same size; the precise length of the vector can be determined later. This means that if one line in your drawing is twice as long as another, it must be made up of twice as many vectors.

DRAWING A SQUARE

Let's work our way through the creation of an actual shape. Below is a simple drawing of a square:



To create a shape table out of this square, we must break each side down into vectors. Since each side is exactly the same size, we can simply say that each is one vector long. This square is made up of four vectors. In breaking this drawing into vectors, we will arbitrarily start with the left side of the square and assume that the computer's "pen" will be moving up when it draws this line. We can therefore assign the left side of the square a vector number of 4. If you'll check the vector chart given earlier, you'll see that this means "move up, while drawing." Since we have

told the computer to move its pen up while drawing this side of the square, it follows with relentless logic that the next vector we ask the computer to draw be on the top of the square, since this vector is but a right angle turn away from the previous vector. Of course, we *could* tell the computer to raise its "pen" (using one of the nondrawing vectors) and move it to another side of the square and draw that unit next, but this is unnecessarily complex. The second vector would therefore have a vector number of 5, for "move right, while drawing."

It doesn't take much thought to see that the right side of the square is next, with the computer's "pen" moving down, which gives the next unit a vector number of 6, for "move down, while drawing." Finally, the computer will draw the bottom of the square, with its "pen" moving to the left, for a vector number of 7 ("move left, while drawing"). This means that we can fully describe this square with four vector numbers, like this:

```
4567
```

If you check these numbers against the vector chart above, you will see that they mean that we drew one unit up, one unit right, one unit down and one unit left. If, at any time in drawing the square, it had become necessary for the computer to move its pen to a new position *without* drawing, we could have used one or more of the vector numbers from 0 to 3 to describe that part of the shape. In this case, however, that was not necessary.

So: do we use the numbers 4567 to describe the square to the computer? Not quite. There is an extra step we must take first, and that extra step requires that we first learn a few things about computer memory and the binary numbering system. If this material turns out to be too difficult for you, you may skip over the next section of this book and go directly to the section later in this chapter entitled "Drawing, Scaling, and Rotating," where we learn how to use a shape table after it has been created. On the other hand, those of you who want to create your own shape tables, or who simply want to understand how a shape table is constructed, had better put on your thinking caps and pay attention to the next section.

BINARY NUMBERS AND SHAPE TABLES

Computers do not think in English or even in BASIC. In fact, computers don't really think at all; rather, they manipulate numbers. All information processed by a computer must be encoded first as numbers—specifically, as a kind of number that we call *binary numbers.*

Binary is a numbering system much like the decimal numbering system that we normally use for counting. In decimal, there are ten possible digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. All decimal numbers are put together from a combination of these digits. In binary, there are only two possible digits: 0 and 1. All possible binary numbers are put together from a combination of these digits. We would count to 5 in binary like this: 0, 1, 10, 11, 100, 101. The binary number 101 is precisely equivalent to the decimal number 5. The binary number 100 is precisely equivalent to the decimal number 4. And so forth.

When any kind of information is placed into a computer, that information is automatically translated into binary numbers. We are not usually aware of this translation, because it happens inside the computer, where we cannot see it. As this translation occurs, the binary numbers are placed one at a time into a series of electronic circuits called *memory,* where they remain while they are being processed by the computer. Each memory circuit in a computer like the Apple II can hold an eight-digit binary number, that is, a binary number containing eight 0's and 1's.

Since it will eventually be necessary to get these numbers back out of memory in some sort of organized manner, each memory circuit is given an identifying number called an *address,* so that a number placed *in* a memory circuit can be found again once it has been stored. On the Apple II series of computers, these addresses are numbers in the range 0 to 65535.

From BASIC, we can examine the contents of these memory addresses using the PEEK function, which we have already used earlier in this book. To PEEK at the contents of a given memory address, place the memory address (or a variable equal to the memory address) in parentheses after the PEEK function; the value of the PEEK function

will be the number that is stored currently at that address. For instance, to learn the contents of address 79, we could write

```
PRINT PEEK(79)
```

If you type this command in the immediate mode and press RETURN, the computer will obligingly print out the contents of memory address 79. You will notice, however, that the number printed out is in decimal, not binary, even though we have clearly stated that the contents of each memory address is a binary number. This is because the Applesoft interpreter, when it responds to our PRINT command, automatically translates the binary number into decimal, on the assumption (probably correct) that most human beings (as opposed to computers) are far more comfortable with decimal numbers than with binary. However, the number stored at memory address 79 is really a binary number. (Actually, it is a series of electrical signals that represent a binary number, but we don't have to worry about this.)

The opposite of the PEEK function is the POKE command, which we have also encountered in passing, earlier in this book. The POKE command will place a number in the computer's memory, at a specified address. We write the POKE command like this:

```
POKE a,n
```

where a is the address into which we wish to place a number, and n is the number we wish to place there. The number n must be a decimal number; the computer will perform the translation to binary automatically. And it must be in the range 0 to 255, because these are the only numbers that can be represented by eight binary digits.

The vector chart in the previous section of this chapter contained the value of the vectors in decimal. Before presenting the shape table to the Applesoft interpreter, however, let's look at the numbers in binary, for reasons that should become apparent in a moment. Here is the vector chart in binary:

000 Up one unit, without drawing
001 Down one unit, without drawing

010 Left one unit, without drawing
011 Right one unit, without drawing
100 Up one unit, while drawing
101 Down one unit, while drawing
110 Left one unit, while drawing
111 Right one unit, while drawing

You'll notice that none of these numbers contains more than three binary digits, while each memory address in the Apple can store an eight-digit binary number. Thus, when we create a shape table and store it in the computer's memory, we will be able to cram two, or even three, of these vectors into a single memory address. And that's where things really start to get sticky.

Our vector-coded version of a square looked like this:

```
4756
```

In binary, the vectors would look like this:

```
100 101 110 111
```

We can cram those four vectors into two eight-digit binary numbers like this:

```
00101100
00111110
```

Notice that we have put the vectors into these numbers from right to left. The rightmost three digits of the first number represent the first vector (100). The three digits before this represent the second vector (101). The leftmost two digits are merely padding.

Similarly, the rightmost three digits of the second number are the third vector (110) and the three digits to the left of these are the fourth vector (111). The leftmost two digits, once again, are padding. If there had been a fifth vector, we would have needed a third number, and we would have crammed the fifth vector into the rightmost three digits of it. If there had been a sixth vector, it would have gone into the three digits to the left of the fifth vector. And so forth.

Later, we will put these two binary numbers into the computer's memory using the POKE command. However,

as noted above, the POKE command requires that we express the number in decimal, rather than binary. How do we translate these numbers into decimal? Well, binary-to-decimal conversions are rather awkward, and we're not going to ask you to perform them in your head, or even on paper. Here, instead, is a short program that will translate binary numbers into decimal:

```
10 INPUT "BINARY NUMBER ";B$
20 IF LEN(B$) = 0 THEN 10
30 DEC = 0
40 FOR I = 1 TO LEN(B$)
50 DEC = DEC * 2
60 IF MID$(B$, I, 1) = "1" THEN DEC = DEC + 1 : GOTO 80
70 IF MID$(B$, I, 1) <> "0" THEN PRINT "NOT A BINARY
NUMBER" : GOTO 10
80 NEXT I
90 PRINT "DECIMAL TRANSLATION: ";DEC
100 GOTO 10
```

When you type and RUN this program, it will prompt you to type a binary number. In response, it will print the decimal translation of the number. If you type the two binary numbers containing our four vectors, you will get these two decimal numbers: 44 and 62.

Now that we have two numbers containing our four vectors, what do we do with them? Are these numbers a shape table? Not exactly. These numbers are a *shape definition.* A shape table is a collection of shape definitions, plus an *index* to those definitions.

Every shape table must begin with an index. The first memory location of the index (and therefore the first location in the shape table) must indicate the number of definitions in the table. If only one definition is in the table, the first address would contain the number 1. If two definitions are in the table, then the first address would contain the number 2. And so forth. Since you are supposed to have only one shape table at a time in the Apple's memory (although there is a way to cheat and get more), you will presumably place all of your shape definitions in one table, and so a program that requires many shapes would have a large number here. You should limit the number of shapes in your table to 255, however.

The second byte of your shape table (*byte* is another word for memory location, that is, for a memory circuit that can hold an eight-digit binary number) will be 0. This will *always* be true.

The next two bytes of the shape table contain the address of the first shape definition relative to the beginning of the table, that is, the number of addresses after the beginning of the table at which we have placed the first shape definition. If the shape table begins at address 768, and the first shape definition is at address 772, then the relative address of this definition is 4, because it begins four addresses after the start of the table. Because this number can potentially be larger than 255 (in a very big shape table), two memory locations are required to store it. (You'll recall that 255 is the biggest number we can place in a single memory location.) Thus, the sixteen-digit binary number that we use to express this address must be broken into two eight-digit chunks. This isn't very easy to do, so here's a BASIC subroutine that will do it for you:

```
1500 SB = INT(AD/256)
1510 FB = AD - SB * 256
1520 RETURN
```

Before you call this subroutine with a GOSUB 1500 instruction, set variable AD equal to the memory address you wish to break into two bytes, in this case, the relative address of the first shape definition. The first byte of the number (that is, the half of the number that will go into the first of the two memory locations) will be returned in variable FB. The second byte of the number will be in variable SB. You may relocate this subroutine to a line number other than 1500, if you wish.

The next bytes contain the relative address of the second definition, if there is one. And the next two bytes contain the relative address of the third definition, if there is a third definition. And so forth. Obviously, you need to include these bytes only if you have more than one definition in your table.

The definitions themselves can go anywhere else in the Apple's memory, as long as you put the proper relative address in the index. However, it is easier to place them immediately after the index (though you may want to leave

a little spare room in the index to add the addresses of additional shapes, should you wish to expand the table later).

A SQUARE TABLE

With this information in mind, we can create a shape table for the square that we created above:

```
1
0
4
0
44
62
```

The first number, 1, tells the Applesoft interpreter that our shape table will contain a single definition. The second number is 0, as it will always be. The third and fourth numbers tell us that the relative address of the first (and only) definition is 4. (You can use the conversion program above to verify that 4 and 0 are the first and second bytes for a relative address of 4, though this should be pretty obvious.) Finally, the fifth and sixth bytes contain the shape definition itself.

We can place this table in the computer's memory via a series of POKE statements. The question arises, however, as to where in the computer's memory we should put it. After all, the Applesoft interpreter is already using a good bit of memory for other purposes, such as storing our programs and the values of our variables. We can't just go poking around in memory arbitrarily. Fortunately, there is a section of memory free between addresses 768 and 975; in fact, this is a common place for storing shape tables, and it provides more than enough space for our current table (which is only six bytes long). So, we can POKE the shape table into place like this:

```
50 POKE 768,1 : POKE 769,0 : POKE 770,4 : POKE 771,0 :
POKE 772,44 : POKE 773,62
```

Finally, we must inform the Applesoft interpreter of the whereabouts of our shape table. We must place the address of the table—that is, the memory address at which the table

begins—in memory locations 232 and 233. To do this, split the address into its first and second bytes, using the program we gave you earlier for doing this. Address 768, split into two bytes, becomes 0 and 3. We inform the interpreter of the whereabouts of the table like this:

```
60 POKE 232,0
70 POKE 233,3
```

Now that we've got the shape table in memory, what can we do with it?

DRAWING, SCALING, AND ROTATING

As mentioned before, we draw a shape on the display with the DRAW command. Before we can draw a shape, however, we must give the interpreter some more information.

For instance, it must know what color the shape is to be drawn in. This is determined the same way all high-resolution colors are determined: with the HCOLOR command. Since we have already studied the HCOLOR command, we need go into no additional detail here.

Then we must tell the interpreter the scale of the drawing, using the SCALE command. The scale is the number of pixels to be used for each vector. (For those of you who skipped the last section, the vector is the unit from which shapes are constructed. Changing the scale of the vector changes the size of the shape.) We use the SCALE command like this:

```
SCALE = n
```

where n is the number of pixels that will be used for each vector. If you want three pixels in every vector, for instance, you would write:

```
SCALE = 3
```

You must also tell the interpreter the angle of rotation for the drawing, with the ROT command. The ROT command rotates a shape around the center of the display. We use the ROT command like this:

```
ROT = n
```

where n is a number between 0 and 255 representing the amount of rotation. If n is 0, the shape will not be rotated at all. If n is 64, or a multiple of 64, the shape will be rotated a full 360 degrees (and thus will not really be rotated at all). The only meaningful rotation values, therefore, are from 0 to 63, with 16 representing a 90-degree rotation, 32 representing 180 degrees, and 48 representing 270 degrees. The number of increments available between these values varies depending on the scale. If the scale is 1, only rotations of 16, 32, and 48 are possible. If the scale is 5 or greater, all ROT values are valid.

Finally: the DRAW command. We use DRAW like this:

```
DRAW n at x,y
```

where n is the number of the shape definition in the table (our square definition, above, would be number 1) and x and y are the horizontal and vertical coordinates, respectively, of the position at which you wish to draw the shape.

To erase a shape (without erasing the background), we can use the XDRAW command, which draws a shape in the *inverse* of the current HCOLOR. Here is a chart of the inverses of high-resolution colors:

COLOR	INVERSE
black	white
white	black
purple	green
orange	blue
green	purple
blue	orange

The syntax for XDRAW is identical to that for DRAW.

Putting all of this together, here is a program that will draw our square on the display in color 3 (white) at coordinates 100,90, erase it, and redraw it at a new rotation, over and over again:

```
10 HGR
20 HCOLOR = 3
30 POKE -16302,0
40 R = 0
50 POKE 768,1 : POKE 769,0 : POKE 770,4 : POKE 771,0 :
POKE 772,44 : POKE 773,62
60 POKE 232,0
70 POKE 233,3
80 SCALE = 30
90 ROT = R
100 DRAW 1 AT 100,90
110 FOR I = 1 TO 500
120 NEXT I
130 XDRAW 1 AT 100,90
140 R = R + 1 : IF R > 255 THEN R = 0
150 GOTO 90
```

Shape creation is probably the most difficult aspect of Apple graphics. We are not going to go into more detail on the subject here, but anyone interested in the subject is urged to experiment with the information supplied in this chapter, and to seek out other books on the subject (see the bibliography in this book). Although the creation of shape tables is difficult, spectacular effects can be achieved with their aid, if you are willing to put an effort into the task.

Suggested Projects

1. Design a shape table for the house that we drew in the last chapter. Design shape tables for any other shapes you wrote programs to draw in the "Suggested Projects" section at the end of the last chapter.

2. Design a shape table for the brick pattern that we created in the last chapter. Design a shape table for the ladder segment that we created in the last chapter. Rewrite the platforms-and-ladders program from the last chapter using shape tables.

BIBLIOGRAPHY

Compute!'s First Book of Apple. Greensboro, N.C.: Compute! Books, 1984. A collection of articles and programs for the Apple II series computers.

Cuellar, Gabriel. *Fancy Programming in Applesoft.* Reston, Va.: Reston, 1983. Advanced programming tricks and techniques for the Applesoft programmer. Includes a set of useful utility programs to assist in both graphics and non-graphics programming.

DeJong, Marvin L. *Apple II Assembly Language.* Indianapolis, Ind.: Sams, 1982. An introduction to Apple machine language programming for the *very* advanced programmer.

Graff, Lois and Goldstein, Larry Joel. *Applesoft Basic for the Apple II & IIe.* Bowie, Md.: Brady, 1984. A tutorial in Applesoft BASIC for beginning programmers.

Poole, Lon, McNiff, Martin, and Cook, Steven. *Apple II User's Guide.* Berkeley, Calif.: Osborne/McGraw Hill, 1983. A comprehensive reference for Apple II users and Applesoft programmers.

INDEX

AD variable, 89
Address (memory), 39, 85
Animation, 26–28
Apple II, 3
Applesoft BASIC, 1
Applications, low-resolution, 31–44
Arcade-style games, 71
ASCII code, 39

Bar charts, drawing, 31–36
BASIC, 3
Binary numbers, 85–90
Black pixels, 54
Boxes, generating, 25, 26
Breakout, 37
Brick-pattern house, drawing, 70–71
Brøderbund, 45–49, 72
Budge, Bill, 48
BX variable, 42
BY variable, 42
Byte, 89
Byte, Willy, 2, 50

Call-back system, 15, 58
Charts, bar, 31–36
Color, changing on screen, 24–25
Color groups, 55–58
Color identification numbers, 52, 54
  in high-resolution mode, 49
  in low-resolution mode, 15
COLOR statements, 15–18, 28
Columns, 6
Commands
  DRAW, 83, 92
  GET, 39
  HCOLOR, 49, 91
  HOME, 7, 18
  HPLOT, 47, 49, 64, 69
  INPUT, 38
  POKE, 18, 86, 88, 90
  ROT, 91–92
  XDRAW, 92

Computer graphics, 1
Concentric rectangles, drawing, 23
Coordinates, 16, 20–21
  even-numbered, 52
  horizontal, 16
  odd-numbered, 52
  vertical, 16
Cursor, 7
Cursor positioning, 7

DATA statements, 74–77
Data Trek, 2, 79
DazzleDraw, 46
Diagonal lines, 20–22
Diamond, drawing, 69
Digital Dimension, 2
DRAW command, 83, 92
Drawing lines, 18–19
Drawing shapes, 22–26
DX variable, 41
DY variable, 41

Electronic Arts, 48, 53
Endpoints, saving, 75–77
Even-numbered coordinates, 52

Fancy shapes, drawing, 69–77, 81
Filled rectangles, drawing, 24, 67
FLASH instruction, 8
Flash mode, 8
Formatting screen, 5–11
FOR-NEXT loops, 18, 21, 24, 64

Games, low-resolution, 36–41
  enhancing, 41–42
GET command, 39
Geyer, Geyer, and Halpern, 78
GR statement, 14–15, 18
Graphics, low-resolution, 13–29
Graphics characters, 6
Graphics display, 13
Group 1 colors, 57
Group 2 colors, 57
Groups, color, 55–58

Hammond, Greg, 2
HCOLOR command, 49, 91
HGR statement, 47, 49
High-resolution, playing with, 49–52
High-resolution graphics, 3–4
High-resolution mode, 45–61
High-resolution shapes, 81–94
HLIN, 19–20, 22
HOME command, 7, 18
Home position, 7
Horizontal coordinates, 16
Horizontal line, drawing, 18–19
Horizontal tab, 7
House sketch, drawing, 69
HPLOT command, 47, 49, 64, 69
HTAB, 7

IBM "clones," 23
Index, 88
INPUT command, 38
Inverse of high-resolution colors, 92
INVERSE statement, 7
Inverse text, 8, 10

Karateka, 59
Krehbiel, Murray, 2

Ladder, drawing, 71
Line graph, 68
Lines, drawing, 18–19
  multiple, 65–66
Line segments, 58–60
Lines, diagonal, 20–22
Little Brick Out, 37
Lode Runner, 72
Loops, FOR-NEXT, 18, 21, 24, 64
Low-resolution applications, 31–44
Low-resolution games, 36–41
  enhancing, 41–42
Low-resolution graphics, 3–4, 13–39

Maze Craze Construction Sets, 79
Memory, 15, 85
Memory location, 39
Method, 52–55
Mode, high-resolution, 45–61

NORMAL instruction, 8
Numbers, binary, 85–90

Odd-numbered horizontal coordinates, 52
One-on-One, 53

PEEK function, 39–40, 85–86
Pictorial elements, 3
Pinball Construction Set, 48
Ping-Pong Game, 37–41
Pixels, 3, 6, 13
  black, 54
  color displays, 55
  making them disappear, 17
  white, 54
PLOT statement, 15–18, 20, 28
PRINT, 5
POKE command, 18, 86–88, 90
POKE – 16302,0, 14, 15, 47
Programs, computer, 5
Projects, 11, 29, 43–44, 60, 78–79, 94

Rectangle, drawing, 22–24, 66
Review of high-resolution mode, 55
RDT command, 91–92
RND function, 25–26, 42
Running, jumping and climbing style arcade game, 71, 73–74

SCALE command, 83, 91
Screen formatting, 5–11
SCRN function, 38
Series of screens with brick-and-ladder patterns, 75
Shape definition, 88
Shape tables, 81, 85–90
Shapes
  drawing, 22–26
  fancy, 69–77, 81
  high-resolution, 81–94
Split screen, 13, 14
Square, drawing, 83–84
Square table, 90–91

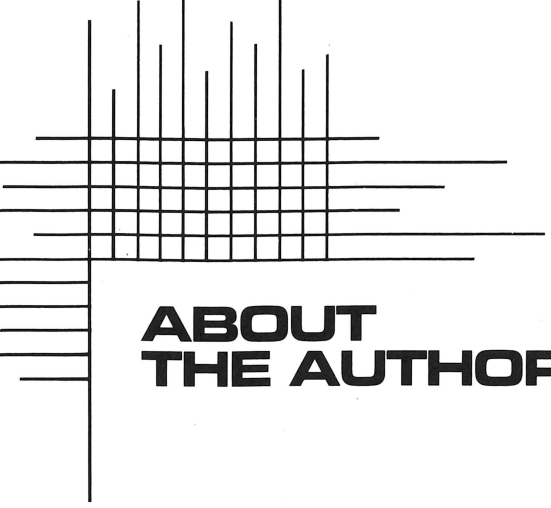
Standard unit, 33–34
Statements
  COLOR, 15–18, 28
  DATA, 74–77
  GR, 14–15, 18
  HGR, 47, 49
  INVERSE, 7
  PLOT, 15–18, 20, 28
String, 5

Text display, 13
TEXT instruction, 14
Text mode, 5
Text window, 14
  getting rid of, 14

Variables
  AD, 89
  BX, 42
  BY, 42
  DX, 41
  DY, 41
Vectors, 82–83
Vertical coordinates, 16
Vertical lines, 18–19, 20
Vertical tab, 7
VLIN, 19–20, 22
VTAB, 7

White pixels, 54

XDRAW command, 92



ABOUT THE AUTHOR

Christopher Lampton is the author of more than twenty books for Franklin Watts, including a number of popular First Book and Impact titles. He has written all the books on computer languages and graphics in Watts' Computer Literacy Skills series.

Chris first became a computer enthusiast when he purchased a Radio Shack computer to use for word processing. He has since acquired eight more computers.

Chris lives in Maryland, right outside Washington, D.C., and has a degree in broadcast journalism. In addition to his books in the areas of science and technology, he has written four science-fiction novels.

ABOUT THE AUTHOR

Christopher Lampton is the author of more than twenty books for Franklin Watts, including a number of popular First Book and Impact titles. He has written all the books on computer languages and graphics in Watts' Computer Literacy Skills series.

Chris first became a computer enthusiast when he purchased a Radio Shack computer to use for word processing. He has since acquired eight more computers.

Chris lives in Maryland, right outside Washington, D.C., and has a degree in broadcast journalism. In addition to his books in the areas of science and technology, he has written four science-fiction novels.

ART DESIGN BY GRANT GEYER,
MARSHALL GEYER, AND JONATHAN HALPERN

PHOTOGRAPH BY IRA SCHULMAN

JACKET DESIGN BY GINGER GILES

PRINTED IN THE UNITED STATES OF
AMERICA BY MOFFA PRESS, INC.

FRANKLIN WATTS' COMPUTER LITERACY SKILLS SERIES

ADVANCED BASIC

BASIC FOR BEGINNERS

COBOL FOR BEGINNERS

FORTH FOR BEGINNERS

FORTRAN FOR BEGINNERS

GRAPHICS AND ANIMATION ON THE APPLE

GRAPHICS AND ANIMATION ON THE ATARI

GRAPHICS AND ANIMATION ON THE COMMODORE 64

GRAPHICS AND ANIMATION ON THE TRS-80

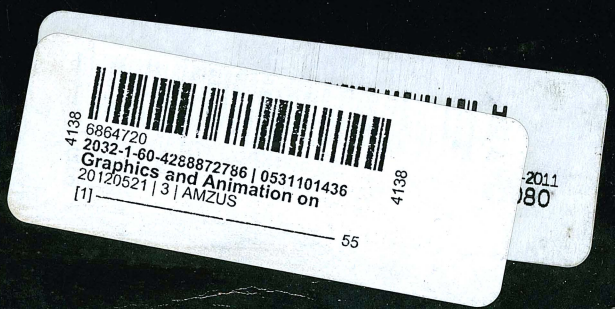
PASCAL FOR BEGINNERS

PILOT FOR BEGINNERS

6502 ASSEMBLY-LANGUAGE PROGRAMMING

Z80 ASSEMBLY-LANGUAGE PROGRAMMING

531-10143-6
BB07


4138 6864720
2032-1-60-4288872786 | 0531101436
Graphics and Animation on
20120521 | 3 | AMZUS
[1] 55
4138
-2011
080

LAMPTON GRAPHICS AND ANIMATION ON THE APPLE WATTS